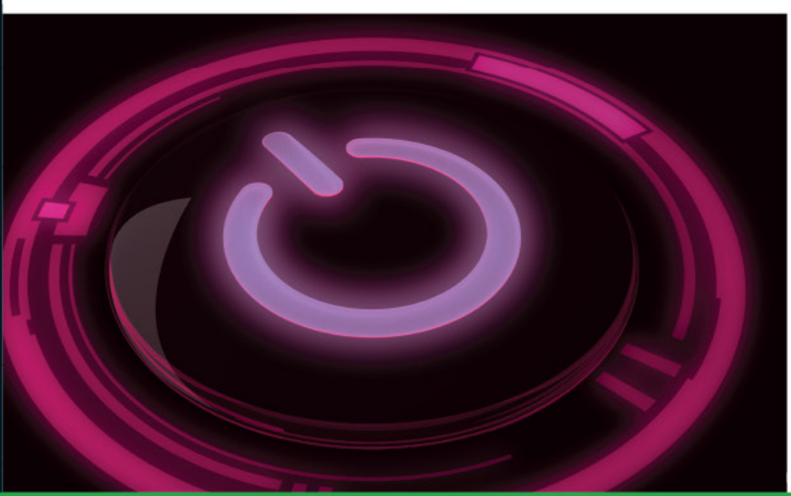
Java, Contai
Beans, Proxi
Databases, W
Server, Jaka
Persistence
API, Spring
Data, Spring
Web MVC, Log



```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@SpringBootConfiguration
@EnableAutoConfiguration
@ComponentScan(excludeFilters = {
      @Filter(type = FilterType.CUSTOM,
                          classes = TypeExcludeFi
      @Filter(type = FilterType.CUSTOM,
                          classes = AutoConfigur
public @interface SpringBootApplication {
  ...
}
```

Spring Boot 3 and Spring Framework 6

Christian Ullenboom

Rheinwerk
Computing

Rheinwerk Computing

The Rheinwerk Computing series from Rheinwerk Publishing offers new and established professionals comprehensive guidance to enrich their skillsets and enhance their career prospects. Our publications are written by leading experts in the fields of programming, administration, security, analytics, and more. Each book is detailed and hands-on to help readers develop essential, practical skills that they can apply to their daily work. For further information, please visit our website: www.rheinwerk-computing.com.

Christian Ullenboom
Java: The Comprehensive Guide
2023, 1126 pages, paperback and e-book
www.rheinwerk-computing.com/5557

Bernd Öggl, Michael Kofler
Docker: Practical Guide for Developers and DevOps Teams
2023, 491 pages, paperback and e-book
www.rheinwerk-computing.com/5650

Sebastian Springer
React: The Comprehensive Guide
2024, approx. 676 pp., paperback and e-book
www.rheinwerk-computing.com/5705

Philip Ackermann
JavaScript: The Comprehensive Guide
2022, 982 pages, paperback and e-book
www.rheinwerk-computing.com/5554

Johannes Ernesti, Peter Kaiser
Python 3: The Comprehensive Guide
2022, 1036 pages, paperback and e-book
www.rheinwerk-computing.com/5566

Christian Ullenboom

Spring Boot 3 and Spring Framework 6

Rheinwerk
Computing

Imprint

This e-book is a publication many contributed to, specifically:

Editor Meagan White
Acquisitions Editor Hareem Shafi
German Edition Editor Almut Poll, Patricia Schiewald
Translation Winema Language Services, Inc.
Copyeditor Julie McNamee
Cover Design Graham Geary
Photo Credit Shutterstock: 110415056/© liewluck; 65652385/© concept w
Layout Design Vera Brauner
Production E-Book Hannah Lane
Typesetting E-Book III-satz, Germany

We hope that you liked this e-book. Please share your feedback with us and read the Service Pages to find out how to contact us.

The Library of Congress Cataloging-in-Publication Control Number for the printed edition is as follows: 2023945877

ISBN 978-1-4932-2475-3 (print)
ISBN 978-1-4932-2476-0 (e-book)
ISBN 978-1-4932-2477-7 (print and e-book)

1st edition 2024
1st German edition published 2023 by Rheinwerk Verlag, Bonn, Germany

Contents

Preface 25

1 Introduction 33

1.1 History of Spring Framework and Your First Spring Project 33
  1.1.1 Tasks of Java Platform, Standard Edition 33
  1.1.2 Enterprise Requirements 33
  1.1.3 Development of Java Enterprise Frameworks 34
  1.1.4 Rod Johnson Develops a Framework 34
  1.1.5 Spring Framework: Many Configurations Are Required 36

1.2 Spring Boot 37
  1.2.1 Spring Boot Versions 38
  1.2.2 Support Period 39
  1.2.3 Alternatives to Spring 40
  1.2.4 Setting Up a Spring Boot Project 43
  1.2.5 Building Spring Projects in Development Environments 48

1.3 Spring Boot Project: Dependencies and Starter 53
  1.3.1 Project Object Model with Parent Project Object Model 54
  1.3.2 Dependencies as Imports 57
  1.3.3 Milestones and the Snapshots Repository 58
  1.3.4 Configuring Annotation Processors 58
  1.3.5 Starter: Dependencies of the Spring Initializr 59

1.4 Getting Started with Configurations and Logging 61
  1.4.1 Turning Off the Banner 62
  1.4.2 Logging API and Simple Logging Facade for Java 62

1.5 Summary 65

2 Containers for Spring-Managed Beans 67

2.1 Spring Container 67
  2.1.1 Start Container 67
  2.1.2 Instantiate SpringApplication 68
  2.1.3 SpringApplicationBuilder * 69
  2.1.4 What main(...) Does and Doesn’t Do 70
  2.1.5 The run(...) Method Returns ConfigurableApplicationContext 70

2.1.6 Context Methods ... 71

2.2 Package Structure of the Date4u Application ... 74

2.3 Pick Up Spring-Managed Beans through Classpath Scanning ... 76

2.3.1 Fill Container with Beans ... 76
2.3.2 @Component ... 77
2.3.3 @Repository, @Service, @Controller ... 80
2.3.4 Control Classpath Scanning More Precisely with @ComponentScan * ... 82

2.4 Interactive Applications with the Spring Shell ... 89

2.4.1 Include a Spring Shell Dependency ... 89
2.4.2 First Interactive Application ... 91
2.4.3 Write Shell Component: @ShellComponent, @ShellMethod ... 91

2.5 Injecting Dependencies ... 95

2.5.1 Build Object Graphs ... 95
2.5.2 Inversion of Control and Dependency Injection ... 97
2.5.3 Injection Types ... 97
2.5.4 PhotoService and PhotoCommands ... 102
2.5.5 Multiple Dependencies ... 105
2.5.6 Behavior in Case of a Missing Component ... 106
2.5.7 Optional Dependency ... 107
2.5.8 Cyclic Dependencies * ... 109
2.5.9 Inject Other Things ... 110

2.6 Configuration Classes and Factory Methods ... 111

2.6.1 What @Component Can't Do ... 111
2.6.2 @Configuration and @Bean ... 111
2.6.3 Parameter Injection of an @Bean Method ... 119
2.6.4 @Configuration Bean and Lite Bean ... 119
2.6.5 InjectionPoint * ... 121
2.6.6 Static @Bean Methods * ... 124
2.6.7 @Import and @ImportSelector * ... 125

2.7 Abstraction and Qualifications ... 127

2.7.1 Bean Name and Alias ... 127
2.7.2 AwtBicubicThumbnail for Thumbnails ... 128
2.7.3 Basic Types ... 131
2.7.4 ObjectProvider ... 140
2.7.5 @Order and @AutoConfigurationOrder * ... 142
2.7.6 Behavior in Selected Inheritance Relationships * ... 144

2.8 Beans Lifecycle ... 146

2.8.1 @DependsOn ... 146
2.8.2 Delayed Initialization (Lazy Initialization) ... 147

2.8.3 Bean Initialization Traditional 149
2.8.4 InitializingBean and DisposableBean * 155
2.8.5 Inheritance of the Lifecycle Methods 155
2.8.6 *Aware Interfaces * 156
2.8.7 BeanPostProcessor * 159
2.8.8 Register Spring-Managed Beans Somewhere Else 163
2.8.9 Hierarchical Contexts * 165
2.8.10 Singleton and Prototype Stateless or Stateful 168

2.9 Annotations from JSR 330, Dependency Injection for Java * 169
2.9.1 Dependency for the JSR 330 Standard Annotation 169
2.9.2 Map JSR 330 Annotations to Spring 170

2.10 Auto-Configuration 170
2.10.1 @Conditional and Condition 171
2.10.2 If, Then: @ConditionalOn* 173
2.10.3 Turn on Spring Debug Logging 180
2.10.4 Controlling Auto-Configurations Individually * 182

2.11 Spring Expression Language 185
2.11.1 ExpressionParser 185
2.11.2 SpEL in the Spring (Boot) API 186

2.12 Summary 190

3 Selected Modules of the Spring Framework 191

3.1 Helper Classes in Spring Framework 191
3.1.1 Components of org.springframework 191

3.2 External Configuration and the Environment 193
3.2.1 Code and External Configuration 193
3.2.2 Environment 194
3.2.3 Inject Values with @Value 195
3.2.4 Get Environment Assignments via @Value and ${…} 197
3.2.5 @Value and Default Values 198
3.2.6 Access to Configurations 200
3.2.7 Nested/Hierarchical Properties 205
3.2.8 Map Special Data Types 211
3.2.9 Relaxed Bindings 216
3.2.10 Property Sources 218
3.2.11 Define Spring Profiles 231
3.2.12 Activate Profiles 235
3.2.13 Spring-Managed Beans Depending on Profile 236

3.3 At the Beginning and End 238
3.3.1 CommandLineRunner and ApplicationRunner 238
3.3.2 At the End of the Application 241
3.3.3 Exit Java Programs with Exit Code * 242

3.4 Event Handling 247
3.4.1 Participating Objects 247
3.4.2 Context Events 248
3.4.3 ApplicationListener 249
3.4.4 @EventListener 250
3.4.5 Methods of the Event Classes 251
3.4.6 Write Event Classes and React to the Events 251
3.4.7 An Event Bus of Type ApplicationEventPublisher 252
3.4.8 Generic Events Using the PayloadApplicationEvent Example 253
3.4.9 Event Transformations 256
3.4.10 Sequences by @Order 256
3.4.11 Filter Events by Conditions 256
3.4.12 Synchronous and Asynchronous Events 257
3.4.13 ApplicationEventMulticaster * 258
3.4.14 Send Events via ApplicationContext 259
3.4.15 Attach Listener to SpringApplication * 259
3.4.16 Listener in spring.factories * 260

3.5 Resource Abstraction with Resource 261
3.5.1 InputStreamSource and Resource 261
3.5.2 Load Resources 263
3.5.3 Inject Resources via @Value 264

3.6 Type Conversion with ConversionService 265
3.6.1 ConversionService 266
3.6.2 DefaultConversionService and ApplicationConversionService 267
3.6.3 ConversionService as Spring-Managed Bean 269
3.6.4 Register Your Own Converters with the ConverterRegistry 270
3.6.5 Printer and Parser 275
3.6.6 FormatterRegistry 277
3.6.7 DataBinder 279

3.7 Internationalization * 283
3.7.1 Possibilities for Internationalization with Java SE 283
3.7.2 MessageSource under Subtypes 284

3.8 Test-Driven Development with Spring Boot 289
3.8.1 Test Related Entries from Spring Initializr 289
3.8.2 Annotation @Test 290
3.8.3 Test Case for the FileSystem Class 290
3.8.4 Test Multitier Applications, Exchange Objects 294

3.8.5 Mocking Framework Mockito 298
3.8.6 @InjectMocks 301
3.8.7 Verify Behavior 302
3.8.8 Testing with ReflectionTestUtils 303
3.8.9 With or Without Spring Support 305
3.8.10 Test Properties 308
3.8.11 @TestPropertySource 309
3.8.12 @ActiveProfiles 310
3.8.13 Appoint Deputy 311
3.8.14 @DirtiesContext 313

3.9 Testing Slices Using a JSON Example * 314
3.9.1 JSON 314
3.9.2 Jackson 315
3.9.3 Write a Java Object in JSON 315
3.9.4 Testing JSON Mappings: @JsonTest 318
3.9.5 Mapping with @JsonComponent 321

3.10 Scheduling * 324
3.10.1 Scheduling Annotations and @EnableScheduling 325
3.10.2 @Scheduled 325
3.10.3 Disadvantages of @Scheduled and Alternatives 328

3.11 Types from org.springframework.*.[lang|util] 328
3.11.1 org.springframework.lang.*Null* Annotations 329
3.11.2 Package org.springframework.util 330
3.11.3 Package org.springframework.data.util 331

3.12 Summary 335

4 Selected Proxies 337

4.1 Proxy Pattern 337
4.1.1 Proxy Deployment in Spring 339
4.1.2 Dynamically Generate Proxies 340

4.2 Caching 343
4.2.1 Optimization through Caching 343
4.2.2 Caching in Spring 344
4.2.3 Component with the @Cacheable Method 345
4.2.4 Use @Cacheable Proxy 346
4.2.5 @Cacheable + Condition 348
4.2.6 @Cacheable + Unless 349
4.2.7 @CachePut 350

4.2.8 @CacheEvict 351
4.2.9 Specify Your Own Key Generators 352
4.2.10 @CacheConfig 358
4.2.11 Cache Implementations 358
4.2.12 Caching with Caffeine 359
4.2.13 Disable the Caching in the Test 360

4.3 Asynchronous Calls 361
4.3.1 @EnableAsync and @Async Methods 362
4.3.2 Example with @Async 362
4.3.3 The CompletableFuture Return Type 365

4.4 TaskExecutor * 369
4.4.1 TaskExecutor Implementations 370
4.4.2 Set Executor and Handle Exceptions 372

4.5 Spring and Bean Validation 374
4.5.1 Parameter Checks 374
4.5.2 Jakarta Bean Validation (JSR 303) 375
4.5.3 Dependency on Spring Boot Starter Validation 375
4.5.4 Photo with Jakarta Bean Validation Annotations 376
4.5.5 Inject and Test a Validator 378
4.5.6 Spring and the Bean Validation Annotations 380
4.5.7 Bean Validation Annotations to Methods 382
4.5.8 Validation Everywhere Using a Configuration Example 382
4.5.9 Test Validation * 383

4.6 Spring Retry * 387
4.6.1 Spring Retry Project 387
4.6.2 @Retryable 389
4.6.3 Fallback with @Recover 392
4.6.4 RetryTemplate 394

4.7 Summary 395

5 Connecting to Relational Databases 397

5.1 Set Up an H2 Database 397
5.1.1 Brief Introduction to the H2 Database 397
5.1.2 Install and Launch H2 398
5.1.3 Connect to the Database via the H2 Console 402
5.1.4 Date4u Database Schema 404

5.2 Realize Database Accesses with Spring 407
5.2.1 Spring Helper 407

5.3 Spring Boot Starter JDBC 408
- 5.3.1 Include the JDBC Starter in the Project Object Model 408
- 5.3.2 Provide JDBC Connection Data 410
- 5.3.3 Inject DataSource or JdbcTemplate 410
- 5.3.4 Connection Pooling 413
- 5.3.5 Log JDBC Accesses 414
- 5.3.6 JDBC org.springframework.jdbc Package and Its Subpackages 415
- 5.3.7 DataAccessException 415
- 5.3.8 Auto-Configuration for DataSource * 418
- 5.3.9 Addressing Multiple Databases 419
- 5.3.10 DataSourceUtils * 421

5.4 JdbcTemplate 422
- 5.4.1 Execute Any SQL: execute(...) 422
- 5.4.2 SQL Updates: update(...) 423
- 5.4.3 Query Individual Values: queryForObject(...) 423
- 5.4.4 Define a Placeholder for a PreparedStatement 424
- 5.4.5 Query Whole Row: queryForMap(...) 425
- 5.4.6 Query Multiple Rows with One Element: queryForList(...) 426
- 5.4.7 Read Multiple Rows and Columns: queryForList(...) 426

5.5 Data Types for Mapping to Results 427
- 5.5.1 RowMapper 428
- 5.5.2 RowCallbackHandler 433
- 5.5.3 ResultSetExtractor 434

5.6 NamedParameterJdbcTemplate 437
- 5.6.1 Type Relationships of *JdbcTemplate 438
- 5.6.2 Methods of the NamedParameterJdbcTemplate 439
- 5.6.3 Pass Values of NamedParameterJdbcTemplate through a Map 439
- 5.6.4 SqlParameterSource 439
- 5.6.5 Access to Underlying Objects * 442

5.7 Batch Operations * 443
- 5.7.1 Batch Methods for JdbcTemplate 443
- 5.7.2 BatchPreparedStatementSetter 444
- 5.7.3 Batch Methods at NamedParameterJdbcTemplate 449
- 5.7.4 SqlParameterSourceUtils 450
- 5.7.5 Configuration Properties 450

5.8 BLOBs and CLOBs * 451
- 5.8.1 SqlLobValue 452
- 5.8.2 LobHandler and DefaultLobHandler 453
- 5.8.3 Read LOBs via AbstractLobStreamingResultSetExtractor 454

5.9 Subpackage org.springframework.jdbc.core.simple * 455
- 5.9.1 SimpleJdbcInsert 456

5.10 Package org.springframework.jdbc.object * 458
5.10.1 MappingSqlQuery 458
5.11 Transactions 460
5.11.1 ACID Principle 460
5.11.2 Local or Global/Distributed Transactions 461
5.11.3 JDBC Transactions: Auto-Commit 461
5.11.4 PlatformTransactionManager 462
5.11.5 TransactionTemplate 466
5.11.6 @Transactional 470
5.12 Summary 472

6 Jakarta Persistence with Spring 473

6.1 World of Objects and Databases 473
6.1.1 Transient and Persistent 474
6.1.2 Mapping Objects to Tables 474
6.1.3 Java Libraries for O/R Mapping 478
6.2 Jakarta Persistence 479
6.2.1 Persistence Provider 480
6.2.2 Jakarta Persistence Provider and JDBC 481
6.2.3 Jakarta Persistence Coverage 481
6.3 Spring Data JPA 482
6.3.1 Include Spring Boot Starter Data JPA 482
6.3.2 Configurations 483
6.4 Jakarta Persistence Entity Bean 485
6.4.1 Develop an Entity Bean Class 486
6.5 Jakarta Persistence API 492
6.5.1 Getting the EntityManager from Spring 492
6.5.2 Search an Entity Bean by Its Key: find(...) 494
6.5.3 find(...) and getReference(...) 496
6.5.4 Query Options with the EntityManager 497
6.6 Jakarta Persistence Query Language (JPQL) 498
6.6.1 JPQL Example with SELECT and FROM 498
6.6.2 Build and Submit JPQL Queries with createQuery(...) 500
6.6.3 Conditions in WHERE 502
6.6.4 Parameterize JPQL Calls 503
6.6.5 JPQL Operators and Functions 506
6.6.6 Order Returns with ORDER BY 510

6.6.7 Projection on Scalar Values ... 511
6.6.8 Aggregate Functions ... 512
6.6.9 Projection on Multiple Values ... 513
6.6.10 Named Declarative Queries (Named Queries) ... 517

6.7 Call Database Functions and Send Native SQL Queries ... 519
6.7.1 Call Database Functions: FUNCTION(...) ... 519
6.7.2 Use createNativeQuery(...) via EntityManager ... 520
6.7.3 @NamedNativeQuery ... 521
6.7.4 @NamedNativeQuery with resultSetMapping ... 522

6.8 Write Access with the EntityManager in Transactions ... 525
6.8.1 Transactional Operations ... 525
6.8.2 persist(...) ... 525
6.8.3 EntityTransaction ... 526
6.8.4 PlatformTransactionManager with JpaTransactionManager ... 527
6.8.5 @Transactional ... 528
6.8.6 Save versus Update ... 529
6.8.7 remove(...) ... 530
6.8.8 Synchronization or Flush ... 530
6.8.9 Query with UPDATE and DELETE ... 531

6.9 Persistence Context and Other Transaction Controls ... 531
6.9.1 Jakarta Persistence API and Database Operations ... 531
6.9.2 States of an Entity Bean ... 532

6.10 Advanced ORM Metadata ... 535
6.10.1 Database-First and Code-First Approaches ... 535
6.10.2 Set the Table Name via @Table ... 536
6.10.3 Change the @Entity Name ... 537
6.10.4 Persistent Attributes ... 537
6.10.5 @Basic and @Transient ... 539
6.10.6 Column Description and @Column ... 539
6.10.7 Entity Bean Data Types ... 542
6.10.8 Map Data Types with AttributeConverter ... 544
6.10.9 Key Identification ... 547
6.10.10 Embedded Types ... 551
6.10.11 Entity Bean Inherits Properties from a Superclass ... 553

6.11 Relationships between Entities ... 555
6.11.1 Supported Associations and Relationship Types ... 555
6.11.2 1:1 Relationship ... 555
6.11.3 Bidirectional Relationships ... 559
6.11.4 1:n Relationship ... 560
6.11.5 n:m Relationships * ... 566

6.12 FetchType: Lazy and Eager Loading 567
6.12.1 Enumeration with FetchType 568
6.12.2 Hibernate Type: PersistentBag 568
6.12.3 1 + N Query Problem: Performance Anti-Pattern 570
6.13 Cascading 573
6.13.1 CascadeType Enumeration Type 574
6.13.2 Set CascadeType 576
6.13.3 cascade=REMOVE versus orphanRemoval=true 577
6.14 Repositories 577
6.14.1 Data Access Layer 578
6.14.2 Methods in the Repository 580
6.14.3 SimpleJpaRepository 582
6.15 Summary 585

7 Spring Data JPA 587

7.1 What Tasks Does Spring Data Perform? 587
7.1.1 For Which Systems Is Spring Data Available? 588
7.2 Spring Data Commons: CrudRepository 589
7.2.1 CrudRepository Type 589
7.2.2 Java Persistence API–Based Repositories 591
7.3 Subtypes of CrudRepository 594
7.3.1 ListCrudRepository 594
7.3.2 Technology-Specific [List]CrudRepository Subtypes 595
7.3.3 Selected Methods via Repository 597
7.4 Paging and Sorting with [List]PagingAndSortingRepository 598
7.4.1 JpaRepository: Subttype of ListPagingAndSortingRepository and ListCrudRepository 599
7.4.2 Sort Type 600
7.4.3 Sort.TypedSort<T> 601
7.4.4 Pageable and PageRequest 602
7.4.5 Sort Paginated Pages 607
7.5 QueryByExampleExecutor * 608
7.5.1 Sample 608
7.5.2 QueryByExampleExecutor 609
7.5.3 Sample into the Example 610
7.5.4 Build ExampleMatcher 611
7.5.5 Ignore Properties 611

7.5.6 Set String Comparison Techniques 612
7.5.7 Set Individual Rules with GenericPropertyMatcher 613
7.5.8 PropertyValueTransformer 613

7.6 Formulate Your Own Queries with @Query 615
7.6.1 @Query Annotation 615
7.6.2 Modifying @Query Operations with @Modifying 617
7.6.3 Fill IN Parameter by Array/Varg/Collection 617
7.6.4 @Query with JPQL Projection 618
7.6.5 Sort and Pageable Parameters 619
7.6.6 Add New Query Methods 620
7.6.7 Queries with Spring Expression Language Expressions 621
7.6.8 Using the @NamedQuery of an Entity Bean 622
7.6.9 @Query Annotation with Native SQL 622

7.7 Stored Procedures * 624
7.7.1 Define a Stored Procedure in H2 625
7.7.2 Calling a Stored Procedure via a Native Query 627
7.7.3 Call a Stored Procedure with @Procedure 627

7.8 Derived Query Methods 628
7.8.1 Individual CRUD Operations via Method Names 628
7.8.2 Structure of the Derived Query Methods 629
7.8.3 Returns from Derived Query Methods 633
7.8.4 Asynchronous Query Methods 633
7.8.5 Streaming Query Methods 634
7.8.6 Advantages and Disadvantages of Derived Query Methods 634

7.9 Criteria API and JpaSpecificationExecutor 635
7.9.1 Criteria API 636
7.9.2 Functional Interface Specification 637
7.9.3 JpaSpecificationExecutor 638
7.9.4 Methods in JpaSpecificationExecutor 639
7.9.5 Specification Implementations 639
7.9.6 Assemble Specification Instances 641
7.9.7 Internals * 642
7.9.8 Metamodel Classes 643
7.9.9 Cons of Using the Criteria API 646

7.10 Alternatives to JDBC Jakarta Persistence 647
7.10.1 Querydsl 648
7.10.2 Spring Data JDBC 655

7.11 Good Design with Repositories 660
7.11.1 Abstractions through the Onion Architecture 660
7.11.2 Think of the Interface Segregation Principle and the Onion! 661
7.11.3 Fragment Interface 662

7.12 Projections 664
- 7.12.1 Perform Projections Yourself 665
- 7.12.2 Projections in Spring Data 665
- 7.12.3 Interface-Based Projection 665
- 7.12.4 Projections with SpEL Expressions 667
- 7.12.5 Projections with Default Methods 668
- 7.12.6 Class-Based Projections 668
- 7.12.7 Dynamic Projections 669

7.13 [Fetchable]FluentQuery * 670

7.14 Auditing * 672
- 7.14.1 Auditing with Spring Data 672
- 7.14.2 Auditing with Spring Data JPA 672
- 7.14.3 AuditorAware for User Information 673
- 7.14.4 Outlook: Spring Data Envers 674

7.15 Incremental Data Migration 675
- 7.15.1 Long Live the Data 676
- 7.15.2 Evolutionary Database Design 676
- 7.15.3 Incremental Data Migration with Flyway 677
- 7.15.4 Flyway in Spring Boot: Migration Scripts 677
- 7.15.5 Migrations in Java Code 680
- 7.15.6 Flyway Migrations outside Spring 681

7.16 Test the Data Access Layer 682
- 7.16.1 What Do We Want to Test? 682
- 7.16.2 Test Slices 683
- 7.16.3 Deploy an In-Memory Test Database 684
- 7.16.4 Assign Connection to the Test Database 684
- 7.16.5 Build Tables with Initialization Scripts 686
- 7.16.6 Testcontainers Project 687
- 7.16.7 Demo Data 689
- 7.16.8 @Sql and @SqlGroup 690
- 7.16.9 TestEntityManager 691

7.17 Summary 691

8 Spring Data for NoSQL Databases 693

8.1 Not Only SQL 693

8.2 MongoDB 694
- 8.2.1 MongoDB: Documents and Collections 694
- 8.2.2 About MongoDB 695

8.2.3 Install and Start the MongoDB Community Server 696
8.2.4 GUI Tools for MongoDB 697
8.2.5 Spring Data MongoDB 698
8.2.6 MongoDB Application Programming Interfaces 699
8.2.7 MongoDB Documents 700
8.2.8 MongoTemplate Class 701
8.2.9 MongoDB Repositories 704
8.2.10 Test MongoDB Programs 706

8.3 Elasticsearch 707
8.3.1 Text Search Is Different 707
8.3.2 Apache Lucene 708
8.3.3 Documents and Fields 708
8.3.4 Index 708
8.3.5 Weaknesses of Apache Lucene 709
8.3.6 Lucene Attachments: Elasticsearch and Apache Solr 710
8.3.7 Install and Launch Elasticsearch 711
8.3.8 Spring Data Elasticsearch 712
8.3.9 Documents 713
8.3.10 ElasticsearchRepository 715
8.3.11 @DataElasticsearchTest 718

8.4 Summary 719

9 Spring Web 721

9.1 Web Server 721
9.1.1 Java Web Server 722
9.1.2 Spring Boot Starter Web 723
9.1.3 Use Other Web Servers * 724
9.1.4 Adjust Port via server.port 724
9.1.5 Serve Static Resources 725
9.1.6 WebJars 726
9.1.7 Transport Layer Security Encryption 726

9.2 Generate Dynamic Content 728
9.2.1 Stone Age: The Common Gateway Interface 728
9.2.2 Servlet Standard 729
9.2.3 Program @WebServlet 730
9.2.4 Weaknesses of Servlets 732

9.3 Spring Web MVC 732
9.3.1 Spring Containers in Web Applications 733
9.3.2 @Controller 734

9.3.3 @RestController ... 735
9.3.4 Controller to [Service] to Repository ... 737

9.4 Hot Code Swapping ... 738
9.4.1 Hot Swapping of the Java Virtual Machine ... 739
9.4.2 Spring Developer Tools ... 739

9.5 HTTP ... 740
9.5.1 HTTP Request and Response ... 740
9.5.2 Set Up an HTTP Client for Testing Applications ... 741

9.6 Request Matching ... 742
9.6.1 @RequestMapping ... 742
9.6.2 @*Mapping ... 743
9.6.3 More General Path Expressions and Path Matchers ... 743
9.6.4 @RequestMapping on Type ... 744

9.7 Send Response ... 745
9.7.1 HttpMessageConverter in the Application ... 745
9.7.2 Format Conversions of Handler Method Returns ... 746
9.7.3 Mapping to JSON Documents ... 746
9.7.4 ResponseEntity = Statuscode + Header + Body ... 752

9.8 Evaluate Request ... 755
9.8.1 Handler Methods with Parameters ... 755
9.8.2 Data Transmission from Client to Controller ... 756
9.8.3 Data Acceptance via Parameters ... 756
9.8.4 Evaluate Query Parameters ... 757
9.8.5 Optional Query Parameters ... 757
9.8.6 Map All Query Parameters ... 758
9.8.7 Evaluate a Path Variable ... 760
9.8.8 MultipartFile ... 763
9.8.9 Evaluate Header ... 765
9.8.10 HttpEntity Subclasses RequestEntity and ResponseEntity * ... 765

9.9 Type Conversion of the Parameters ... 766
9.9.1 YearMonth Converter Example ... 766
9.9.2 @DateTimeFormat and @NumberFormat ... 767
9.9.3 Map Query Parameters and Form Data to a Bean ... 768
9.9.4 Register Your Own Type Converters ... 772
9.9.5 URI Template Pattern with Regular Expressions ... 774

9.10 Exception Handling and Error Message ... 775
9.10.1 Map Exceptions to Status Codes Yourself ... 775
9.10.2 Escalates ... 775
9.10.3 Configuration Properties server.error.* ... 777
9.10.4 Exception Resolver ... 780

9.10.5 Map Exception to Status Code with @ResponseStatus 780
9.10.6 ResponseStatusException 781
9.10.7 Local Controller Exception Handling with @ExceptionHandler 783
9.10.8 RFC 7807: “Problem Details for HTTP APIs” 785
9.10.9 Global Controller Exception Handling with Controller Advice 787

9.11 RESTful API 789
9.11.1 Principles behind REST 789
9.11.2 Implement REST Endpoints for Profiles 795
9.11.3 Data Transfer Objects 797
9.11.4 Best Practice: Don’t Deliver a Complete Collection at Once 800
9.11.5 GET and DELETE on Individual Resources 800
9.11.6 POST and PUT with @RequestBody 802
9.11.7 UriComponents 805

9.12 Asynchronous Web Requests * 806
9.12.1 Long Queries Block the Worker Thread 806
9.12.2 Write Asynchronously to the Output 807
9.12.3 A Handler Method Returns Callable 808
9.12.4 WebAsyncTask 808
9.12.5 A Handler Method Returns DeferredResult 809
9.12.6 StreamingResponseBody 809

9.13 Spring Data Web Support 811
9.13.1 Loading from the Repository 811
9.13.2 Pageable and Sort as Parameter Types 812
9.13.3 Return Type Page 814
9.13.4 Querydsl Predicate as a Parameter Type 815

9.14 Documentation of a RESTful API with OpenAPI 817
9.14.1 Description of a RESTful API 817
9.14.2 OpenAPI Specification 817
9.14.3 Where Does the OpenAPI Document Come From? 819
9.14.4 OpenAPI with Spring 819
9.14.5 springdoc-openapi 820
9.14.6 Better Documentation with OpenAPI Annotations 822
9.14.7 Generate Java Code from an OpenAPI Document 822
9.14.8 Spring REST Docs 824

9.15 Testing the Web Layer 825
9.15.1 Test QuoteRestController 825
9.15.2 Annotation @WebMvcTest 827
9.15.3 Writing a Test Method with MockMvc 827
9.15.4 Test REST Endpoints with the Server 830
9.15.5 WebTestClient 831

9.16 Best Practices When Using a RESTful API 832
9.16.1 Jakarta Bean Validation 832
9.16.2 Resource ID 833
9.16.3 Map Data Transfer Objects 834
9.16.4 Versioning of RESTful Web Services 836

9.17 Secure Web Applications with Spring Security 837
9.17.1 The Importance of Spring Security 837
9.17.2 Dependency on Spring Boot Starter Security 838
9.17.3 Authentication 838
9.17.4 SecurityContext and SecurityContextHolder 839
9.17.5 AuthenticationManager 842
9.17.6 SpringBootWebSecurityConfiguration 843
9.17.7 AuthenticationManager and ProviderManager 846
9.17.8 UserDetailsService Interface 847
9.17.9 Spring-Managed Bean PasswordEncoder 852
9.17.10 BasicAuthenticationFilter 856
9.17.11 Access to the User 857
9.17.12 Authorization and the Cookie 858
9.17.13 Token-Based Authentication with JSON Web Tokens 859

9.18 Consume RESTful Web Services 867
9.18.1 Classes for Addressing HTTP Endpoints 867
9.18.2 A WebClient Example 868
9.18.3 Declarative Web Service Clients 873

9.19 Summary 877

10 Logging and Monitoring 879

10.1 Logging 879
10.1.1 Why Create a Protocol? 879
10.1.2 Log Group 880

10.2 Logging Implementation 880
10.2.1 Conversion to Log4j 2 881
10.2.2 Logging Pattern Layout 881
10.2.3 Change the Logging Configuration 883
10.2.4 Banner 883
10.2.5 Logging at Start Time 884
10.2.6 Testing Written Log Message 884

10.3 Monitor Applications with Spring Boot Actuator 885
10.3.1 Determine the Health Status via the Actuator 886

10.3.2 Activation of the Endpoints 887
10.3.3 Info Supplements 889
10.3.4 Parameters and JSON Returns 889
10.3.5 New Actuator Endpoints 891
10.3.6 Am I Healthy? 892
10.3.7 HealthIndicator 893
10.3.8 Metrics 893

10.4 Micrometer and Prometheus 894
10.4.1 Micrometer 895
10.4.2 Prometheus: The Software 898

10.5 Summary 900

11 Build and Deployment 901

11.1 Package and Run Spring Boot Programs 901
11.1.1 Deployment Options 901
11.1.2 Launching a Spring Boot Program via Maven 902
11.1.3 Packing a Spring Boot Program into a Java Archive 902
11.1.4 spring-boot-maven-plugin 903

11.2 Spring Applications in the OCI Container 904
11.2.1 Container 904
11.2.2 Install and Use Docker 906
11.2.3 H2 Start, Stop, Port Forwarding, and Data Volumes 908
11.2.4 Prepare a Spring Boot Docker Application 910
11.2.5 Docker Compose 913
11.2.6 Terminate Applications with an Actuator Endpoint 914

11.3 Summary 915

Appendices 917

A Migration from Spring Boot 2 to Spring Boot 3 917

B The Author 923

Index 925
Service Pages I
Legal Notes II

Preface

What Is Spring Boot?

Spring Boot, the leading Java enterprise framework, provides numerous benefits to the developer community. It's user-friendly, simplifies microservices development, and, when developed correctly, can achieve scalability, making it ideal for building robust enterprise applications.

This book covers all the essential features of the widely used Java framework, up to its latest major release, *Spring Boot 3*, which was launched in November 2022. It has been developed over several years to offer a comprehensive overview of the framework.

Prior Knowledge and Target Group

As a widely adopted framework in the Java community, Spring has evolved into a de facto standard in recent years. This book is tailored for individuals interested in creating enterprise applications that incorporate database connectivity and web services. It caters to both novices and seasoned developers, with a plethora of practical examples included. However, a solid grasp of Java and proficiency in tools such as Maven are required prerequisites, along with a fundamental understanding of relational databases.

Date4u Demo Application, Tasks, and Solutions

This book, in conjunction with a reliable development environment, equips you with the skills to create Spring programs. However, simply reading about a new framework is insufficient to become proficient. To learn a programming language or framework thoroughly, you must practice and become fluent, treating it like a foreign language where the Spring framework is the vocabulary.

To illustrate the central components of an enterprise application, this book employs an amusing example of a dating application for unicorns. The demonstration showcases the interaction of components, storage of profile data in a database, and access via web services. You are guided through each step of the application with tasks scattered throughout the text for practice. Each task is accompanied by a suggested solution to facilitate sequential reading. The *Date4u* application, which exemplifies the concepts discussed in the book, is downloadable at *https://rheinwerk-computing.com/5764/*.

Chapter Organization

The book is divided into the following chapters:

Chapter 1, "Introduction," introduces the *Spring Framework* and the "extension" *Spring Boot*. The chapter first shows how to create a project and then explains the structure of the Maven project object model (POM) file, starters and dependencies, and configuration and logging.

Chapter 2, "Containers for Spring-Managed Beans," focuses on the Spring context. Step by step, a mesh of Spring-managed beans is built from a single class to store and load images in the file system.

Managing Spring-managed beans is a core task of the Spring framework, but there are many other features addressed in **Chapter 3**, "Selected Modules of the Spring Framework," including external configuration, event handling, conversion classes, and various utility classes.

Chapter 4, "Selected Proxies," continues with a discussion of special "rings" that wrap around components and thus take over caching, asynchronous calls, or validation. This offers the advantage that certain tasks can be shifted to the framework so that your code becomes slenderer.

The first four chapters focus on the Spring container and helper classes, but not on storage technologies. That changes in **Chapter 5**, "Connecting to Relational Databases," in which a database management system is prepared and SQL queries to the database are demonstrated.

Writing to databases using the Java Database Connectivity (JDBC) application programming interface (API) is simple and direct, but inconvenient. Therefore, **Chapter 6**, "Jakarta Persistence with Spring," introduces object-relational mapping.

Data access layers are necessary in every major application. **Chapter 7**, "Spring Data JPA," shows the possible uses of data access layers by using the family member *Spring Data JPA* as an example. It's easy to write repositories with Spring Data, and many queries and data transformations are realized by the framework.

Relational databases are flanked by nonrelational database systems, which are also excellently integrated into the Spring Data family. **Chapter 8**, "Spring Data for NoSQL Databases," uses MongoDB and Elasticsearch to show how Spring Data concepts translate to the NoSQL world.

After describing an important part of the infrastructure with data storage, we continue with applications of HTTP in **Chapter 9**, "Spring Web." There, the focus is on RESTful web services and HTTP clients.

With the knowledge acquired so far, you could operate an application, but it's important to also be able to look into the application from the outside to check whether the "operating temperature" is right. **Chapter 10**, "Logging and Monitoring," introduces the Spring Boot Actuator project as a way of releasing data from an application that can then be retrieved and visualized by external tools.

The fact that a program runs in the development environment and doesn't fail in the test cases is good, but it's only half the battle because a program must be rolled out. For this purpose, **Chapter 11**, "Build and Deployment," introduces Open Container Initiative (OCI) containers (aka Docker), among other things.

Spring Boot 3 is new, and it will be a while before teams move to Java 17. Those developing applications on Java 8 will find some tips on migrating an application from Spring Boot 2 to Spring Boot 3 in **Appendix A**, "Migrating from Spring Boot 2 to Spring Boot 3."

The * Sections

The Spring framework encompasses many intricacies that can be overwhelming for beginners and novices. To help distinguish between essential and nonessential information, some section headings are marked with an asterisk (*), indicating that they can be skipped without sacrificing important concepts required for subsequent chapters.

Which Java Version Is Used in the Book?

Spring Boot 3 necessitates Java 17, which comes with long-term support. This implies that runtime vendors provide extensive support for many years to ensure that the release remains up to date for an extended period. Nevertheless, Spring Boot 3 can also work with more recent versions of Java, such as Java 21.

Required Software

To execute Java programs, a Java virtual machine (JVM) is required. Initially, obtaining a runtime environment was a simple task. The runtime environment originated from Sun Microsystems and was subsequently adopted by Oracle after acquiring Sun. However, the present situation is much more perplexing. Several institutions compile the *OpenJDK*, which comprises hundreds of thousands of lines of C(++) and Java source code, and package it into a distribution known as the Java runtime environment. The most well-known Java runtime environments include the following:

- Eclipse Adoptium (*https://adoptium.net*)
- Amazon Corretto (*https://aws.amazon.com/corretto*)
- Red Hat build of OpenJDK (*https://developers.redhat.com/products/openjdk/overview*)

There are others as well, such as those from Azul Systems or BellSoft. To learn with this book, you can use a distribution of your choice; if you're not sure, you'll do fine with Adoptium.

Development Environment

Java source code is just text, so in principle, a simple text editor is enough. However, you can't expect great productivity with an editor such as Notepad. Because modern development environments support us in many ways—color highlighting of keywords, automatic code completion, intelligent error correction, inserting code blocks, visualization of states in the debugger, and many more—it's advisable to use a full integrated development environment (IDE). Three popular IDEs are *IntelliJ*, *Eclipse*, and *Visual Studio Code*. On the other hand, *Apache NetBeans* is losing more and more ground.

As with the Java runtime environments, you can choose which development environment you want to use. Eclipse, NetBeans, and Visual Studio Code are free and open source. *IntelliJ IDEA Community Edition* is also free and open source, but the more powerful *IntelliJ IDEA Ultimate Edition* is not. The Ultimate version of IntelliJ is certainly the most powerful Java IDE on the market, and its support for Spring is exemplary and unmatched by other IDEs.

Conventions

Several conventions are used in this book, as follows:

- Newly introduced terms are *italicized*, and the index (usually) points to that exact location. Furthermore, *file names*, *HTTP addresses, executable program names*, *program options,* and *file extensions* (*.txt*) appear in italics.
- User interface terms are in **Bold**.

Additional rules apply to code and listings:

- Listings, methods, and other program elements are set in `non-proportional` font.
- In some places, a curved arrow ↩ has been placed after a listing line as a special character that marks the line break, meaning the code from the next line still belongs to the previous one. Method names and constructors are always followed by a pair of parentheses to distinguish the methods/constructors from object/class variables. Thus, with the notation `System.out` and `System.gc()`, it's clear that the former is a static variable and the latter a static method.
- If a method or constructor has parameters, they are enclosed in parentheses: `run(String... args)`. Usually, the parameter name is omitted, so it's called briefly: `run(String...)`. If the return type is relevant in the context, it's included, like this: `ConfigurableApplicationContext run(String... args)` or `ConfigurableApplicationContext run(String...)`. If a method or constructor has a parameter list without being relevant currently, it's abbreviated with an ellipsis, for example, "A `run(…)`

method starts the container." Thus, an empty pair of parentheses means that a method or constructor has no parameter list.

- To make the parameter list short and yet precise, there are partial statements in the text like `getProperty(String key[, String def])`, which is an abbreviation for `getProperty(String key)` and `getProperty(String key, String def)`. The ellipsis can also be found in other places when the implementation or screen outputs aren't required for understanding.
- To specify a group of methods, the * identifier symbolizes a placeholder. For example, `print*(…)` refers to the methods `println(…)`, `print(…)`, and `printf(…)`. The context indicates which methods are meant.
- Long package names are sometimes abbreviated, so that `org.springframework.data.jpa.repository` becomes `o.s.d.j.r`, for example.
- If annotations occur in the text, it's redundant to write "`@Value`-annotation", but the `@` sign in the text allows the reader to grasp it more quickly.

To indicate compiler errors or runtime errors in the program code, the line contains a ☠. Thus, it can be read at first glance that the line either isn't compiled or throws an exception at runtime due to a programming error. An example is shown here:

```
SpringApplication.run( WebApplication.class, null )
// java.lang.IllegalArgumentException: Args must not be null ☠
```

Programs are sometimes called from the command line (synonym: *console*, *shell*). Because each command-line program has its own prompt sequence, it's generically symbolized by a `$` here in the book. Our inputs are set in bold, as shown in the following example:

```
$ java -version
openjdk version "17.0.2" 2022-01-18
OpenJDK Runtime Environment (build 17.0.2+8-86)
OpenJDK 64-bit Server VM (build 17.0.2+8-86, mixed mode, sharing)
```

In places where the Windows command line is explicitly meant, I use the prompt character `>`:

```
> ver
Microsoft Windows [version 10.0.19041.388]
```

Program Listings

As a rule, only the relevant code excerpts are included in the listings so as not to exceed the scope of the book. Thus, package names and import declarations usually don't appear. Most of the source code is either linked as a snippet or is part of the demo project, so the name of the file or the URL to the code snippet is mentioned in the listing caption, like this:

```
class ABC { }
```

Listing 1 ABC.java

The source code shown is in the file *ABC.java*. The package specification isn't mentioned because no two type declarations are the same in different packages in the projects. For other resources, such as XML or HTML documents, the same pattern applies:

```
<xml …>
```

Listing 2 pom.xml

The complete source code is available via download (see *https://rheinwerk-computing.com/5764/*).

Some listings aren't part of the Date4u project and are linked, like this:

```
# http://web.archive.org/web/20111224041840/http://ww…
```

Listing 3 https://gist.github.com/ullenboom/80cdb6c7435980c14b887a85d4b667ec

Because a code block usually evolves, the addition of "extension" indicates that it has changed; the new code is usually set in bold:

```
class ABC {
  public final static String INTRO = "When will it start?";
}
```

Listing 4 ABC Extension

Download and Online Information

You'll find every bit of source code for the Date4u software as well as updates to the book at *https://rheinwerk-computing.com/5764/*. All program chunks from Date4u are free of rights and can be taken over and modified in your own programs without permission. The ZIP file from the download contains the following components:

- **main**
  This is the main Date4u application with services for uploading and downloading images and database access via Jakarta Persistence. Spring Shell enables interactive use with shell commands, such as listing profiles. RESTful web services make profile data accessible from the outside. This is a Maven project that can be integrated into any Java IDE.

- **unicorns**
  This is a directory with 120 different pictures of unicorns. The file names are *unicorn001.jpg* to *unicorn120.jpg*. A ZIP file with the images is also available at *https://github.com/ullenboom/120-unicorn-photos*.
- **h2-2022-06-13**
  This directory contains a snapshot of the current version of the database management system called H2. The *bin* directory contains the shell scripts *h2.[bat|sh]*, which can be used to start the database directly, provided that the JVM is included in the search path.
- **unicorn-database**
  This directory contains the file *unicorns.sql* with a SQL script for the database management system H2 to initialize the example database. You can also find it at *https://tinyurl.com/4fu3hwu4*.
- **product-database**
  This is a Spring boot program with an example of accessing the NoSQL database MongoDB. It's a Maven project.
- **chat-messages**
  This Spring boot program includes an example of Elasticsearch, which is a Maven project.

About the Author

Christian Ullenboom typed his first lines of code into the C64 at the age of 10. After many years of assembler programming and early BASIC extensions, his journey led him to the island of Java after studying computer science and psychology. Vacations to Python, JavaScript, TypeScript, and Kotlin haven't been able to rid him of the savantism yet.

For more than 25 years, Christian has been an enthusiastic software architect, Java trainer (*www.tutego.com*), and IT specialists trainer. His training activities have resulted in several technical books, including *Java: The Comprehensive Guide*. The books have been bringing readers closer to the Java programming language for many years. For his special commitment, Sun (now Oracle) awarded Christian the status of *Java Champion* in 2005 as a personality who has made a special contribution to Java.

Christian also shares his knowledge and experience via learning videos at *https://tutego.learnworlds.com/*. These videos also cover an introduction to Java and Spring Boot. Occasionally, he posts developer news and other nonsense on the Mastodon instance *https://mas.to/@ullenboom*.

Christian Ullenboom has his roots in Sonsbeck on the Lower Rhine, Germany.

Feedback

No matter how carefully we've gone through the chapters, with more than 900 pages, some inconsistencies are likely, just as any software has purely statistical errors.[1] If you have comments, suggestions, corrections, or questions about specific points or general didactics, don't hesitate to contact me at *ullenboom@gmail.com*. I'm always grateful for suggestions, praise, and corrections—also for cheese and licorice.

For now, I wish you lots of spring feelings with Spring Boot.

---

[1] Statistically, there are about 1 to 25 errors per 1,000 lines of code during development. For references, see *https://stackoverflow.com/questions/2898571/basis-for-claim-that-the-number-of-bugs-per-line-of-code-is-constant-regardless*.

Chapter 1
Introduction

Spring is a comprehensive framework, so we want to approach it step by step. The first thing you need to do is make sure that the project has the right dependencies. At the end of this chapter, we will have built your first Spring boot project that can be compiled and launched. In the following chapters, we'll continue to expand this project and fill it with life.

1.1 History of Spring Framework and Your First Spring Project

To begin, let's take a brief look at the history of the Spring Framework and how Spring Boot came to be. We'll start with the question of why *Java Platform, Standard Edition* (Java SE) often isn't sufficient for enterprise applications.

1.1.1 Tasks of Java Platform, Standard Edition

Java SE provides a solid foundation for creating a wide range of applications. The platform includes essential features and functionalities such as input/output operations, data structures, networking capabilities, date and time calculations, concurrency support, and basic security features. In addition, Java SE includes two libraries, AWT and Swing, that provide graphical user interface components. While Java SE offers a wide range of functionalities, larger tasks may require the use of additional libraries.

1.1.2 Enterprise Requirements

Developing complex business applications often requires a wide range of additional functionalities and technologies beyond what Java SE provides. These additional requirements may include the need for RESTful web services, dynamic web page generation, database access (both relational and nonrelational), messaging systems, application monitoring, emailing, and more.

While Java SE provides a solid foundation for building applications, it doesn't include all the necessary technologies and tools required for enterprise-level development. To meet these additional requirements, an extended technology stack is needed.

1.1.3 Development of Java Enterprise Frameworks

Sun Microsystems released the first version of Java in the mid-1990s. At that time, the internet was becoming popular, and the demand for dynamically generated web content was increasing. Sun Microsystems quickly responded to this demand by introducing the *Servlet API* in 1996. This application programming interface (API) allowed developers to write programs that could be executed on web servers and respond to HTTP requests.

The Servlet standard defines an API with which you can ensure your own programs reside in the web server and can be addressed. The Servlet API still plays an important role today and is implemented, for example, by the popular *Tomcat* servlet container. While the Servlet API is only about reacting to HTTP requests, Sun then introduced the *Java Platform, Enterprise Edition* at the end of 1999, that is, just under five years after the release of Java SE. This specification is implemented by an *application server*, which can administer reusable components called *Enterprise JavaBeans* (now *Jakarta Enterprise Beans*) on the server side.

The *Java 2 Platform, Enterprise Edition (J2EE)*, had an interesting start, but there were numerous problems with it. For example, the first generation of application servers were memory intensive and had a long startup time. Another problem arose because the business logic was very closely interwoven with the Java Enterprise API. The use of *invasive technologies* in this context was troublesome because, for example, any J2EE interfaces had to be implemented when implementing the business logic. Another problem was that applications were difficult to test, usually only within the running application server.

The long startup time also meant that automated execution of tests or deployments took a long time. In addition, the Java language didn't yet have as many features as it has today; for example, annotations weren't introduced until Java 5. Spring uses a strongly declarative approach to build applications as flexibly as possible. If you go back to 1999, there were no annotations yet. Instead, much had to be coded in XML files, which was the only way to express something declaratively at the time.

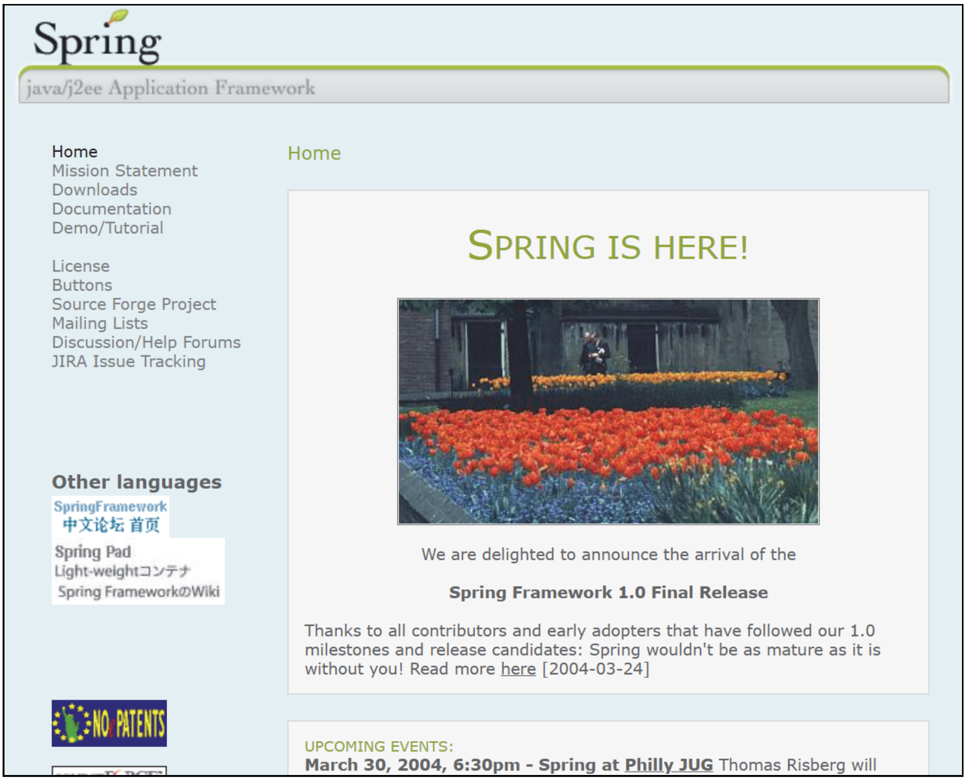
Another problem with J2EE was that the specification was a good start, but it didn't cover many cases. The manufacturers of the application servers were quick to offer their own solution for these problems; the only dilemma was that the application then no longer ran on every application server, of course, but was tied to a technical implementation, creating a vendor lock.

1.1.4 Rod Johnson Develops a Framework

These shortcomings were well known, and one person in particular—Rod Johnson—wanted to improve the situation. He developed a new framework called *Interface 21* for his book *J2EE Design and Development* (Wrox, 2002). This framework was already quite extensive at the time, with around 30,000 lines of code, and was pompously described as the “interface for the 21st century.” The name “Spring,” under which we know it today, came a little later. The interesting thing is that Rod Johnson didn't plan to write a framework: at the time, he saw himself more as a book author who wanted to show how modern enterprise applications could be developed using J2EE.

Like other authors in this field, Johnson offered the source code of his framework for download on the publisher's website.[1] In the Wrox forum, the framework attracted the interest of Jürgen Höller and Yann Caroff in particular. They discussed the code in the Wrox forum and motivated Rod Johnson to put the code on SourceForge. In early 2003, the source code moved from a Wrox publisher download to a project on SourceForge. Caroff also suggested a new name, *Spring*, which is basically as full-bodied as "Interface 21"; it was supposed to be the new spring for J2EE applications. You can still find the old artifacts and contributions at *https://sourceforge.net/projects/springframework/*.

A little later, Johnson, Höller, and Caroff founded a company called Interface 21. In March 2004, the first release of the Spring Framework appeared (see Figure 1.1).


Spring
java/j2ee Application Framework
Home
Mission Statement
Downloads
Documentation
Demo/Tutorial
License
Buttons
Source Forge Project
Mailing Lists
Discussion/Help Forums
JIRA Issue Tracking
Other languages
SpringFramework
中文论坛 首页
Spring Pad
Light-weightコンテナ
Spring FrameworkのWiki
NOePATENTS
Home

SPRING IS HERE!

We are delighted to announce the arrival of the

Spring Framework 1.0 Final Release

Thanks to all contributors and early adopters that have followed our 1.0 milestones and release candidates: Spring wouldn't be as mature as it is without you! Read more here [2004-03-24]

UPCOMING EVENTS:

March 30, 2004, 6:30pm - Spring at Philly JUG Thomas Risberg will

Figure 1.1 2004 Spring Web Page Announcing the 1.0 Release[2]

---

[1] The book was published by Wrox Publishing, which became insolvent in 2003. Some titles were taken over by John Wiley & Sons.

[2] The Web Archive has archived the old website at *https://web.archive.org/web/20040330131500/http://www.springframework.org:80/*. Figuratively speaking, enterprise applications are blooming again with the Spring Framework. Twenty years ago, the SourceForge hosting platform for open-source software was what GitHub is today.

For a few years, Interface 21 earned money by consulting and implementing software projects. In 2007, Benchmark Capital invested $10 million in Interface 21, which was later renamed SpringSource. VMware acquired SpringSource in August 2009 for about $420 million. Johnson had to be pleased; in mid-2012, he left VMware and withdrew from Spring development.

To sum it up, the Spring Framework is now a serious business endeavor and no longer just a personal project or an offshoot of a book. Thanks to the support of VMware, there are dedicated developers working on improving the framework, and it has become a reliable foundation for creating enterprise-level Java applications.

1.1.5 Spring Framework: Many Configurations Are Required

The Spring Framework was developed in response to the heavyweight J2EE applications that required complex configurations.[3] The project was initiated around 2002, and in March 2004, the first release, Spring Framework 1.0, was made available. The early versions of the Spring Framework were also complex to set up, but they offered a lighter weight alternative to J2EE. Examples of Spring Framework 1.0 can be found, for example, on GitHub page *https://github.com/ullenboom/spring-framework-1.0-samples*. There are quite a few projects, including the fairly well-known *petclinic*. The example, in a modernized form, still exists today.

In the mirrored directory *https://github.com/ullenboom/spring-framework-1.0-samples/tree/main/petclinic/war/WEB-INF*, several XML files can be made out; this is typical for the former configuration. We have to keep in mind that there were no annotations in the Java language at that time, and whoever wanted to work declaratively had no alternative but to configure the application via XML files.

Let’s look at a (somewhat reformatted) snippet of *applicationContext-jdbc.xml* to see how components were configured under Spring:

```
<bean id="dataSource"
  class="org.springframework.jdbc.datasource.DriverManagerDataSource">
  <property name="driverClassName">
    <value>${jdbc.driverClassName}</value>
  </property>
  <property name="url">
    <value>${jdbc.url}</value>
  </property>
  <property name="username">
    <value>${jdbc.username}</value>
  </property>
```

---

[3] Back then, people liked to generate the necessary J2EE XML descriptors from JavaDoc: *https://xdoclet.sourceforge.net/xdoclet/index.html*.

```
  <property name="password">
    <value>${jdbc.password}</value>
  </property>
</bean>
<bean id="transactionManager" class= ↩
  "org.springframework.jdbc.datasource.DataSourceTransactionManager">
  <property name="dataSource">
    <ref local="dataSource" />
  </property>
</bean>
```

The code excerpt demonstrates how to define and configure *Spring-managed beans* in the Spring context. The `<bean>` element specifies the bean ID, class, name, and type. The nested `<property>` element later leads to the invocation of the bean's setter. For instance, database connections require details such as the Java Database Connectivity (JDBC) driver's name, JDBC URL, username, and password, which are declared through an extensive series of property declarations. The `<ref local="dataSource"/>` element indicates an injection where the Spring-managed bean, `transactionManager`, requires a `dataSource`, for instance.

As XML files grew in size, configuring applications became increasingly complicated. However, because there were no other alternatives at the time, configuring applications with XML files was the only option. Fortunately, Spring Framework 2.5 supports Java annotations and configuration, enabling developers to define Spring components in Java rather than XML. This simplified the configuration process considerably, providing a more manageable and streamlined approach to application development.

While the Spring Framework's flexibility in configuration is an advantage, it also creates a problem. The extensive configurability of Spring programs necessitates configuring everything before the programs can be executed. Setting up a new Spring Framework project from scratch can be time-consuming and challenging, requiring a significant ramp-up time. This issue required a solution, which came in the form of Spring Boot. Spring Boot streamlines the configuration process and reduces ramp-up time, making it easier to set up and run Spring Framework applications.

1.2 Spring Boot

In 2012, Mike Youngstrom suggested on the mailing list that Spring applications should actually be much easier to configure. Here is a small snippet of the message (*https://jira.spring.io/browse/SPR-9888*):

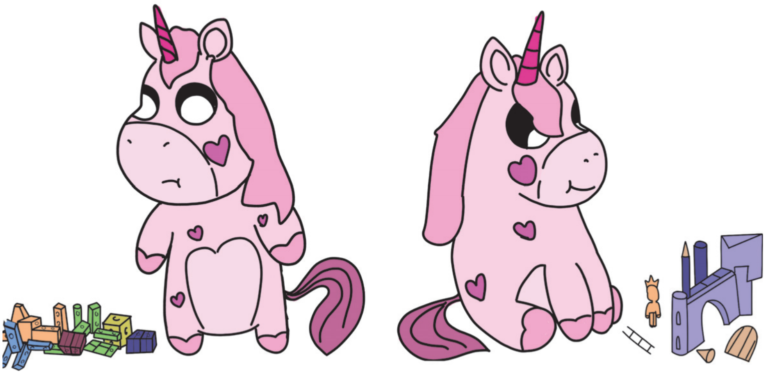
> *I think that Spring's web application architecture can be significantly simplified if it were to provide tools and a reference architecture that leveraged the Spring component and configuration model from top to bottom. Embedding and unifying the*

> *configuration of those common web container services within a Spring container bootstrapped from a simple main() method.*

The idea of not needing a servlet container to deploy a Spring application was also tempting to others. Instead, the Spring application would have its own `main(…)` method and could optionally launch an embedded servlet container itself, especially for web applications.

Today, when we refer to Spring applications, we mostly mean Spring applications configured via Spring Boot. The Spring Framework provides the actual capabilities, with certain components added, such as testing and database access, among others. However, all of these components require configuration before they can be used. This is where Spring Boot comes in. It acts as a ring around the framework and creates an appropriate default configuration for all the components, allowing developers to start working right away without worrying about the configuration details. Merely by having a class or setting a few properties, Spring Boot magically provides complex services in context, which is called *auto-configuration*.[4]

Currently, Spring Boot has become the de facto standard for building Spring applications. Therefore, from now on, we'll use the term "Spring" to refer to the combination of Spring Boot and Spring Framework. When we refer to technologies specifically from Spring Framework or Spring Boot, we'll make that clear. Today, Spring applications are typically configured automatically through Spring Boot.



1.2.1 Spring Boot Versions

The developments on Spring Boot started in 2013. The first release, *Spring Boot 1.0*, appeared in 2014, followed about four years later by the second major release, *Spring Boot 2.0*. The last minor release, *Spring Boot 2.7*, has been around since the end of May

---

[4] I'll adopt the hyphenated spelling.

2022. *Spring Boot 3* appeared in November 2022 and is based on *Spring Framework 6*, which was completed shortly before.

There are two major lines in the releases so far (see Figure 1.2). The first version of Spring Boot has long been deprecated, and the version 2 branch is the one that is most widely used today. It's not possible to predict as of today how quickly Spring Boot version 3 will be adopted.

Updates appear regularly. While there used to be a strong focus on feature releases, as in the early years of Java SE and Jakarta Enterprise Edition (Jakarta EE), the Spring Boot team has now also moved away from releasing by feature, to releasing in regular increments. With the Spring Framework, about six to eight weeks pass from one release to the next. That's why the intervals between Spring Boot releases 2.2, 2.3 . . . are about the same. Additionally, following major or minor updates, regular patches are consistently released. This can be seen quite well by the small dots in the circles in Figure 1.2; in the early days of version 2, there were still patches for version 1, until around the middle of 2019, when the version 1 series was phased out; since then, it has only continued with the version 2 release. From version 2.3 on, there are updates every month.

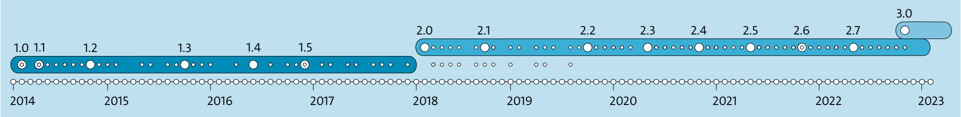


Figure 1.2 Release Data of the Spring Boot Versions

Regularly updating the Spring Boot version is crucial as it defines a set of sub-versions for libraries that are automatically updated by the Spring team. Therefore, we don't have to worry about updating these versions ourselves. It's essential to keep the Spring Boot version up-to-date to take advantage of all the sub-updates.

1.2.2 Support Period

The Spring Framework and also Spring Boot are open source and are under Apache license 2.0. Nevertheless, there is commercial support from VMware; the advantages are documented at *https://tanzu.vmware.com/spring-runtime*. Support is appropriate when an older version needs to continue to be supported, as is apparent for Spring Boot 2, which will no longer receive support after November 18, 2023. Customers with commercial support currently have until February 18, 2025, to make the switch. Many organizations won't be able to switch to Java 17 for Spring Boot 3, making commercial support the only safe option. An overview of support time frames is provided at *https://spring.io/projects/spring-boot#support* (see Figure 1.3).

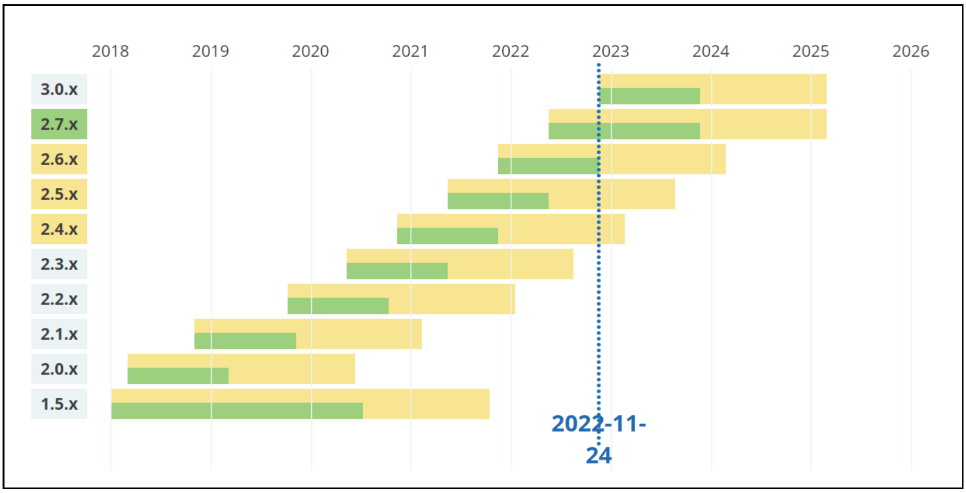


Figure 1.3 Support Periods of the Different Spring Boot Versions

The yellow shading on the web page shows the commercial support; the green shading shows the bug fixes and updates of the open-source versions.

1.2.3 Alternatives to Spring

The Spring Framework was initially created in response to the complexity and inconvenience of J2EE at the time. However, it has been more than a decade since then, and it's worth considering whether the criticisms still hold true for the current Java Enterprise Edition. Additionally, it's worth exploring whether there are viable alternatives to the Spring Framework and whether Spring itself has kept up with modern developments in software development.

J2EE later became *Java Enterprise Edition* (Java EE), and after some wrangling (Oracle didn't want to give up the right to the name "Java"), then *Jakarta EE*. Jakarta EE is a comprehensive continuation of the original J2EE and has little in common with the original. Today, Jakarta EE is a good model for developing large enterprise Java applications.

In addition to the aforementioned alternatives to Spring, there are quite a few other enterprise frameworks on the market:

- Dropwizard, *www.dropwizard.io* (V1, July 2016)
- Micronaut, *https://micronaut.io* (V1, May 2018)
- Helidon, *https://helidon.io* (V1, February 2019)
- Quarkus, *https://quarkus.io* (V1, March 2019)

Note

Mike Youngstrom himself uses Dropwizard as a model for Spring Boot.

To summarize, in principle, there are exciting alternatives to Spring. Jakarta EE also fulfills all the important requirements of an enterprise framework today so that you can write business applications with it. So, why still use Spring?

A comparison between Jakarta EE and Spring isn't really possible, but a comparison between a Jakarta EE application server and a Spring application makes more sense. We must not forget that Jakarta EE is basically just a bundle of about 20 sub-specifications that an application server fully implements. The parts include the following:

- Jakarta Bean Validation
- Jakarta Concurrency
- Jakarta Enterprise Beans
- Jakarta Enterprise Web Services
- Jakarta Expression Language
- Jakarta Faces
- Jakarta Mail
- Jakarta Persistence
- Jakarta RESTful Web Services
- Jakarta Server Pages (JSP)
- Jakarta Servlet

There are many more as well, so we have a whole set of part specifications. Spring also uses a lot of these parts, for example, *Jakarta Bean Validation*. Spring uses this good standard and reference implementation because there's no reason for the Spring makers to develop something entirely new. The same is true with the *Mail API*—there is no reason to disdain the Mail API because sending and receiving email isn't exactly trivial. That is, in the Spring universe, you don't just develop new standards when the problem is already solved. Another example is the *Jakarta Persistence API*. O/R mapping with the Jakarta Persistence API is complex enough, and there are mature implementations.

That's why the comparison is difficult because if we look at the Spring framework, it's more of an integration framework that encompasses different technologies. If you look for a difference between Jakarta EE and Spring, it's the *application server*. The application server is started and runs along all day long, while applications are added, removed, and swapped out all the time. An application server contains a web server that is also always running. A Spring application usually contains an embedded web server, which means that Spring applications usually aren't deployed in a container, but contain all server components.

Jakarta EE and its application servers aren't really competitors for the Spring Framework, but there are other developments where alternative frameworks are a bit ahead. Modern frameworks, such as *Quarkus*, are optimized for microservices and use *native compilation*. This means that there is no traditional Java virtual machine that is started, reads in the byte code at runtime, and translates it into machine code in several iterations at runtime. The translation is done before the execution and is therefore also called *ahead-of-time compilation* (*AOT compilation*). Oracle offers such a compiler, and the technology is part of *GraalVM Native Image*. This results in a directly executable file at the end, for example, an *.exe* file in Windows, which we can start by double-clicking on it. Native Java applications have radically reduced startup time and lowered memory consumption. Spring has taken a long time to support native compilation; during that time, other solutions such as Quarkus have found fans. But since Spring Framework 6 and Spring Boot 3, native compilation has also moved in. In addition, other features favor Spring.

The big advantage of the Spring framework is the capability to integrate everything. Let's compare it again with Jakarta EE: While in the Jakarta environment, we have exactly one API for a task (e.g., email or object-relational mapping [ORM]) with different implementations. In Spring, however, it's the other way around: there are different APIs and therefore also different implementations. This is well illustrated by the example of an object-relational mapper (O/R mapper). The Jakarta Persistence defines an API for object-relational mapping, but there are definitely alternative approaches, for example, with *jOOQ*, *Querydsl*, or *Spring Data JDBC*, there is more than just one O/R mapper.

While it's true that a Spring application can use all Jakarta technologies, Spring goes much further. For example, while only Jakarta Faces and JSP are standardized in Jakarta EE, any template engine can be used in the Spring universe. These include *Thymeleaf*, *FreeMarker*, *Velocity*, and, of course, *Jakarta Faces*. One of Spring's central approaches is to be noninvasive, that is, not to mix Spring-specific data types with business logic. In the Spring environment, we rarely have any connection to an actual technology; instead, it's often encapsulated by an abstraction of Spring.

Spring also offers solutions for which there is nothing in the Jakarta EE standard. If you look at the projects on the Spring site, you'll find many solutions in the cloud environment that can be used to easily develop and manage microservices.

Spring is still one of the best ways to build enterprise applications today, and different surveys show that. JRebel regularly surveys which technologies are used in software development (see Figure 1.4); Spring Boot is at the top of the list (actually up 12 percentage points from last year).[5]

The combination of Spring Boot and the Tomcat servlet container is what most teams use today for modern Java enterprise applications. Of course, other Java enterprise frameworks are also used, but the share isn't high at the moment.

---

[5] *www.jrebel.com/system/files/jrebel-2022-java-developer-productivity-report.pdf*

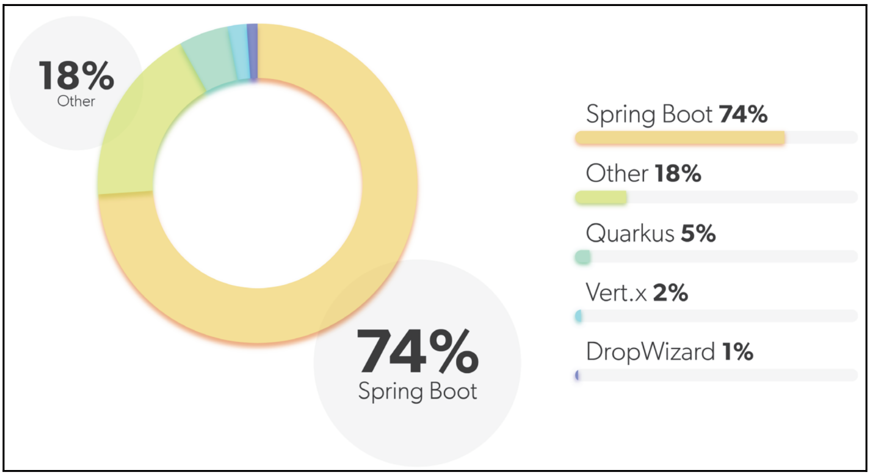


Figure 1.4 Spring Boot Penetration according to a Survey by JRebel

Note

The progression of the development of Java applications is somewhat peculiar: Initially, Java programs were executed within a container—the enterprise application server. Then, containers were deemed obsolete, and applications had to contain all components. However, in the present scenario, there is a trend of moving back toward some form of container, where applications are typically hosted in the cloud per the serverless model.

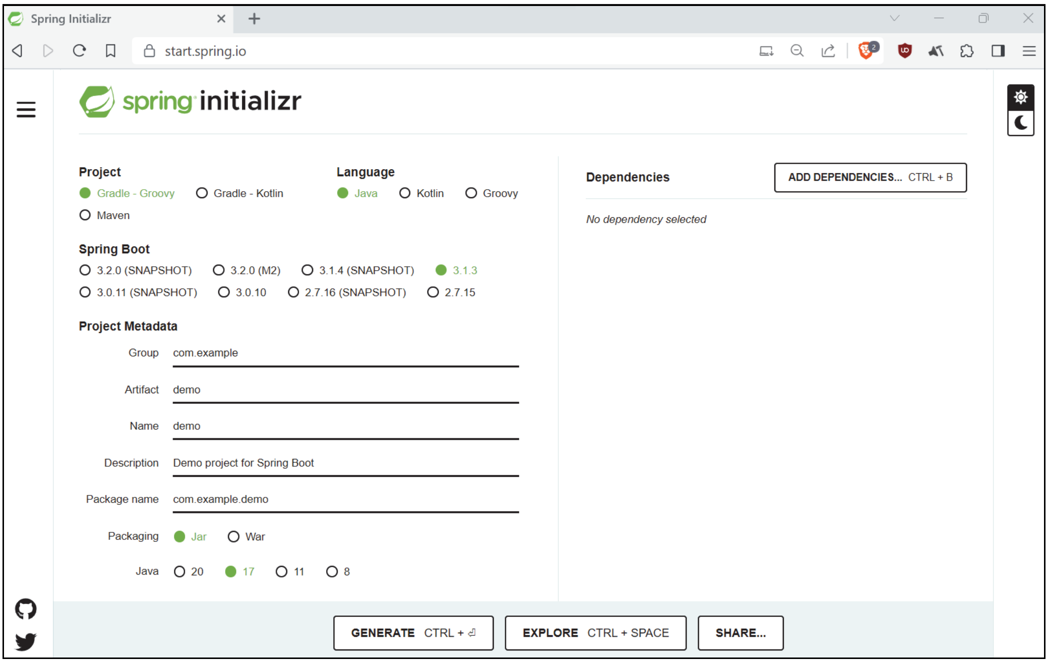
1.2.4 Setting Up a Spring Boot Project

There are several ways to build new Spring Boot projects. A Spring Boot project is essentially a Java project that includes a few classes in its classpath. Nowadays, dependencies aren't manually added to the classpath; instead, build tools such as *Maven* or *Gradle* are commonly used to manage them. While building a Spring Boot project manually isn't complicated, it's advisable to use a project generator.

Spring Initializr

With the *Initializr*, there is a web-based service that we can use directly in the browser. In most modern development environments, this is also directly integrated via dialogs. Let's take a look at the Initializr at *https://start.spring.io/* (see Figure 1.5).

We can choose whether a Maven or a Gradle project as desired; in this case, we choose **Maven**. Next, we can set the programming language. **Language** options include **Java**, **Kotlin**, and **Groovy**, with Spring's reference documentation featuring examples in both Java and Kotlin, highlighting the importance of Kotlin in backend development. Java is the default language, and we'll keep it that way.



Project
● Gradle - Groovy ○ Gradle - Kotlin ○ Maven

Language
● Java ○ Kotlin ○ Groovy

Spring Boot
○ 3.2.0 (SNAPSHOT) ○ 3.2.0 (M2) ○ 3.1.4 (SNAPSHOT) ● 3.1.3
○ 3.0.11 (SNAPSHOT) ○ 3.0.10 ○ 2.7.16 (SNAPSHOT) ○ 2.7.15

Project Metadata
Group com.example
Artifact demo
Name demo
Description Demo project for Spring Boot
Package name com.example.demo
Packaging ● Jar ○ War
Java ○ 20 ● 17 ○ 11 ○ 8

Dependencies
No dependency selected

Figure 1.5 Spring Initializr Website

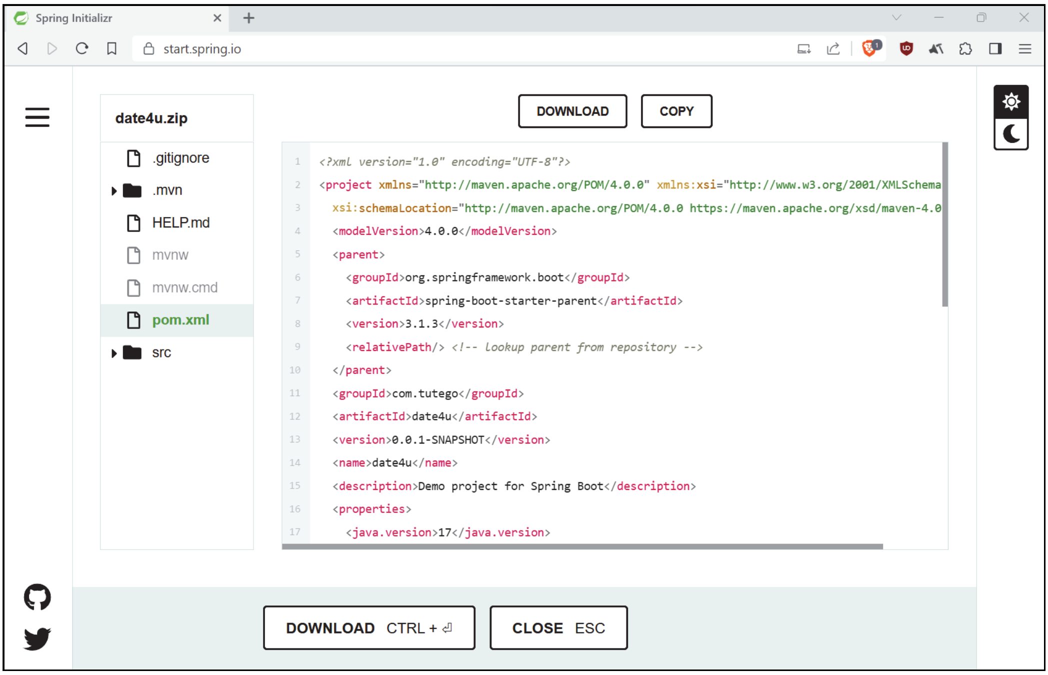
Next, we select the version number of Spring Boot (because there are updates every month, the options will be more current on the website than the screenshot here). Then, we can enter the usual project metadata, which is typical for Maven. This has nothing to do with the Spring project at first.

To build a project, we fill in the following input fields: enter “com.tutego” for **Group** and “date4u” for **Artifact**. The **Name** field is prefilled as **date4u**, and **Description** is prefilled as **Demo project for Spring Boot**. The **Package name**, which is automatically generated from the **Group** and the **Artifact**, fits so far.

Next, we look at **Packaging**: Should the application be packaged as a **JAR** (Java archive) (the default) or generated as a web application (**WAR**)? Web applications can be deployed later in a servlet container such as Tomcat. In addition, the Java version must be specified; Spring Boot 3 requires at least Java 17.

The **Dependencies** section on the right allows for the inclusion of Spring-supported projects, rather than any arbitrary Java library. For example, you could say “I want to do something with web development” or “something with a tool like Lombok” so that there are setters and getters automatically via a compiler hack. The details are later automatically entered into the project object model (POM) file (or in the case of Gradle, into a Gradle file) via the Initializr. Dependencies can be removed by using the minus sign.

Selecting **Explore there** shows a preview into the project as it would be generated (see Figure 1.6).



```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>3.1.3</version>
    <relativePath/> <!-- lookup parent from repository -->
  </parent>
  <groupId>com.tutego</groupId>
  <artifactId>date4u</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <name>date4u</name>
  <description>Demo project for Spring Boot</description>
  <properties>
    <java.version>17</java.version>
```

Figure 1.6 A First Look at the Project

If you unfold the structure tree at *src*, you see the typical *default directory layout* of Maven: *src/main/java*, *src/main/resources*, and *src/test/java*, but no *src/test/resources* by default. This must be added manually later.

In addition, other files can be found: a main class with the `main(…)` method, an empty *application.properties* file for configuration, an automatically generated test class, and, of course, as already shown, the POM file. There is also a *HELP.md* file that contains compact documentation about the dependencies. In addition, *.gitignore* is generated because developers today usually use Git for version control.

On the web page with the file tree, there is a **Download** button at the bottom (see Figure 1.6). Clicking on it opens a ZIP file. On the main page, the button is called **Generate** (refer to Figure 1.5). In other words, the Spring Initializr is essentially a special web service that provides a source code archive with the corresponding project.

Once you've downloaded and unpacked the ZIP archive, the result is a regular Maven project that can be opened in any modern development environment. (For example, if you're using the free *IntelliJ IDEA Community Edition*, you'll have to do so as a Maven project at this point because that edition has no support for Spring Boot projects.) However, the project can now also be built and executed on the command line because the Initializr uses the *Maven wrapper* to generate a kind of local Maven installation. The installation can be tested with `mvnw -version`:

```
$ ./mvnw -version
Apache Maven 3.8.6 (84538c9988a25aec085021c365c560670ad80f63)
Maven home: …\.m2\wrapper\dists\apache-maven-3.8.6-bin\
1ks0nkde5v1pk9vtc31i9d0lcd\apache-maven-3.
8.6
Java version: 17, vendor: Oracle Corporation, runtime: ...
\jdk-17
Default locale: en_DE, platform encoding: Cp1252
OS name: "windows 10", version: "10.0", arch: "amd64", family: "windows"
```

Tip

Maven requires a set `JAVA_HOME` environment variable. If the procedure described causes any problems, it's probably due to the environment variable not being set. However, Maven also displays this in an error message.

A plug-in is already added to the POM file, which can also be used to run the Spring program. After entering the command `mvnw spring-boot:run`, all phases are processed. This means that the project is first compiled and then executed.

```
$ mvnw spring-boot:run

[INFO] Scanning for projects…
[INFO]
[INFO] ------------------------< com.tutego:date4u >----------------------
[INFO] Building date4u 0.0.1-SNAPSHOT
[INFO] -------------------------------[ jar ]-----------------------------
[INFO]
[INFO] >>> spring-boot-maven-plugin:3.0.0:run (default-cli) > test-
compile @ date4u >>>
[INFO]
[INFO] --- maven-resources-plugin:3.3.0:resources (default-resources) @ date4u -
--
[INFO] Copying 1 resource
[INFO] Copying 0 resource
[INFO]
[INFO] --- maven-compiler-plugin:3.10.1:compile (default-compile) @ date4u ---
[INFO] Changes detected - recompiling the module!
[INFO] Compiling 1 source file to …\target\classes
[INFO]
[INFO] --- maven-resources-plugin:3.3.0:testResources (default-
testResources) @ date4u ---
[INFO] skip non existing resourceDirectory …\src\test\resources
[INFO]
```

```
[INFO] --- maven-compiler-plugin:3.10.1:testCompile (default-
testCompile) @ date4u ---
[INFO] Changes detected - recompiling the module!
[INFO] Compiling 1 source file to …\target\test-classes
[INFO]
[INFO] <<< spring-boot-maven-plugin:3.0.0:run (default-cli) < test-
compile @ date4u <<<
[INFO]
[INFO]
[INFO] --- spring-boot-maven-plugin:3.0.0:run (default-cli) @ date4u ---
[INFO] Attaching agents: []

  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::                (v3.1.3)

2023-11-28T18:38:14.846+01:00  INFO 61404 --- [
           main] com.tutego.date4u.Date4uApplication      : Starting Date4uAppli
cation using Java 17 with PID 61404 (…\target\classes started by Christian in …)
2023-11-28T18:38:14.850+01:00  INFO 61404 --- [
           main] com.tutego.date4u.Date4uApplication      : No active profile se
t, falling back to 1 default profile: "default"
2023-11-28T18:38:15.252+01:00  INFO 61404 --- [
           main] com.tutego.date4u.Date4uApplication      : Started Date4uApplic
ation in 0.71 seconds (process running for 0.998)
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  3.103 s
[INFO] Finished at: 2023-11-28T18:38:15+01:00
[INFO] ------------------------------------------------------------------------
```

In the console, we see the result: the Spring application starts up and shuts down again. Because the application doesn't contain a server part, there's nothing else for the Spring application to do, and it exits automatically.

Using the Maven wrapper has two key benefits: First, project recipients can run the project immediately without needing their own local Maven installation. Second, the local version of Maven provided by the wrapper doesn't conflict with any other global Maven installation. This can be advantageous in achieving a *reliable build*, where the software can be built not only today but also 10 years from now. When the software provides its own Maven, the build environment becomes a closed system that works without any external dependencies. The Maven wrapper downloads all the necessary components from the Maven Central server, and the only requirement is an installed JDK. Additional information is available on the official website at *https://maven.apache.org/wrapper/index.html*.

1.2.5 Building Spring Projects in Development Environments

Using the Initializr or the *Spring Boot CLI* to generate the project is fine. However, development environments help build Spring projects quickly and easily. Additional tooling helps especially with listing the components (e.g., the HTTP endpoints), with certain validity checks of method names, or with querying databases.

There are corresponding plug-ins for all modern development environments:

- **Eclipse IDE**
  The "official IDE" of VMware, the *Spring Tool Suite* (*https://spring.io/tools*), is available for the *Eclipse IDE*. This extends an Eclipse installation with corresponding plug-ins and provides a convenient way to create projects.
- **IntelliJ IDE**
  Only the IntelliJ IDEA Ultimate Edition comes with Spring support. Unfortunately, those who use the *IntelliJ IDEA Community Edition* won't find support for Spring projects, but eventually, it's also what distinguishes the free version from the expensive version. There are some plug-ins on the market that extend the free IntelliJ IDEA Community Edition a bit. However, these are proprietary and only the last option if you can't use the *IntelliJ IDEA Ultimate Edition*. There are occasional screenshots of the IntelliJ IDEA Ultimate Edition in this book.
- **Visual Studio Code**
  Visual Studio Code is an editor that is becoming more and more popular. A VMware plug-in can be installed, and then you can create and run new Spring Boot projects effortlessly and comfortably within Visual Studio Code.
- **Apache NetBeans**
  Fans of NetBeans can also install a plug-in (*https://plugins.netbeans.apache.org/catalogue/?id=4*), which is the second most popular plug-in ever.

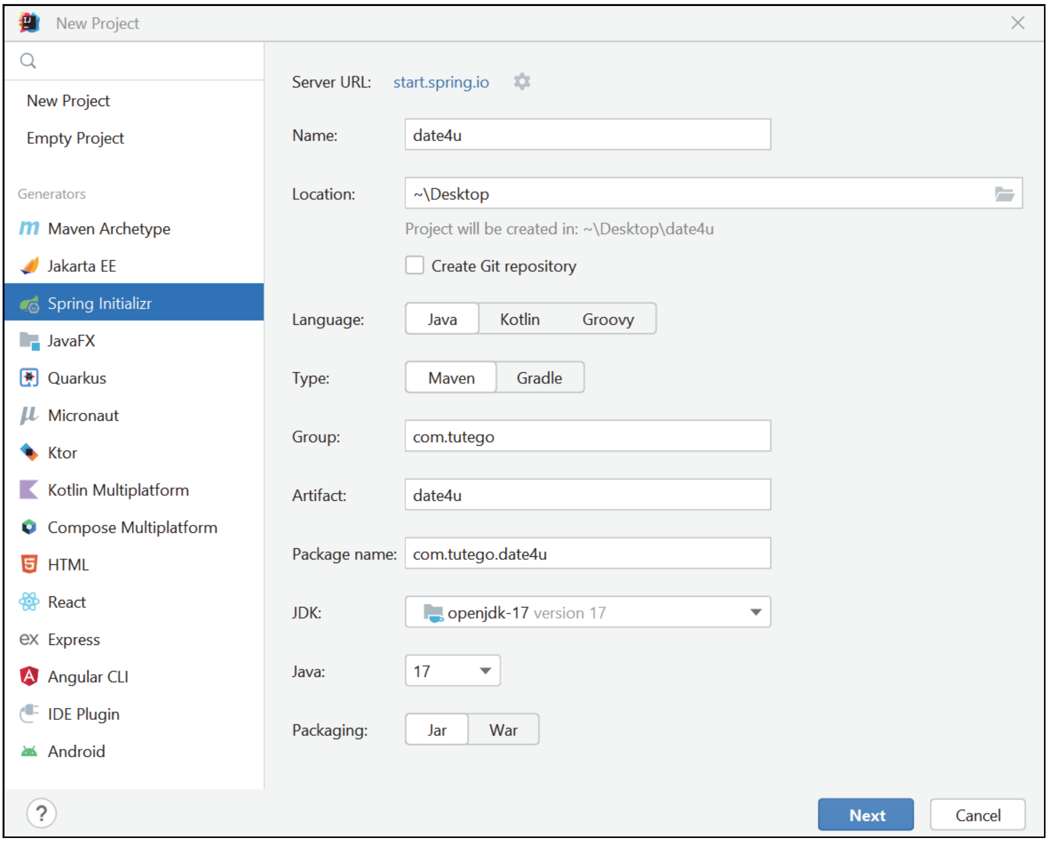
The next sections show how the different IDEs help to build a Spring Boot project without additional dependencies. If you already know this, you can skip these sections and build a project on your own with the following Maven coordinates:

- Group ID: `com.tutego`
- Artifact ID: `date4u`
- Package: `com.tutego.date4u`

IntelliJ IDEA Ultimate Edition

Basically, you can implement Spring Boot projects with all major development environments, but IntelliJ support is the best.

To build a new project, we go to the **File** menu and then click on **New** and **Project**. From there, we can create various project types. On the left side, select **Spring Initializr** (see Figure 1.7).

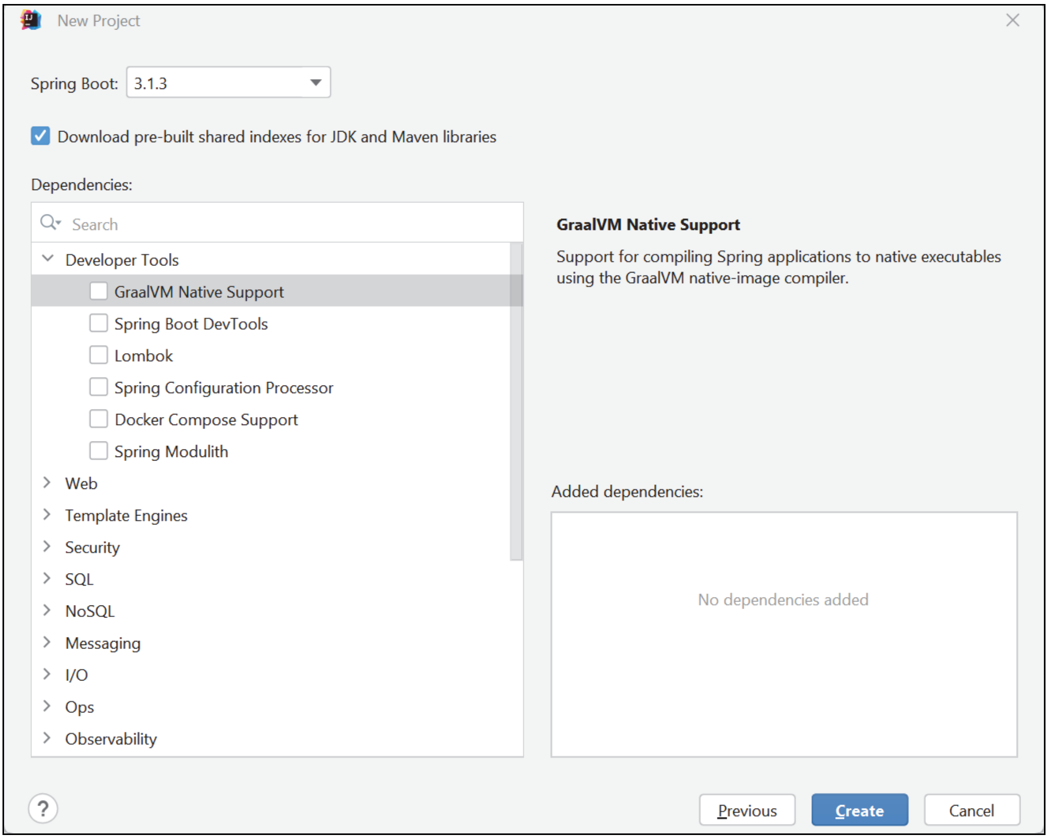

New Project
New Project
Empty Project
Generators
Maven Archetype
Jakarta EE
Spring Initializr
JavaFX
Quarkus
Micronaut
Ktor
Kotlin Multiplatform
Compose Multiplatform
HTML
React
Express
Angular CLI
IDE Plugin
Android
?
Next
Cancel

Server URL: start.spring.io
Name: date4u
Location: ~\Desktop
Project will be created in: ~\Desktop\date4u
☐ Create Git repository
Language: Java | Kotlin | Groovy
Type: Maven | Gradle
Group: com.tutego
Artifact: date4u
Package name: com.tutego.date4u
JDK: openjdk-17 version 17
Java: 17
Packaging: Jar | War

Figure 1.7 Dialog for Creating a New Project

The dialog is strongly related to the Initializr web page. Valid data is entered, so you could just continue with **Next**, but we want to adjust some values.

With the programming language choices of Java, Kotlin, and Groovy, we stay with **Java**. We also select **Maven** not **Gradle**. For the **Name**, enter “date4u”. The destination folder in the **Location** field can be customized. For the **Group** field, enter “com.tutego”, and for **Artifact**, enter “date4u”. The **Package name** should be “com.tutego.date4u”. Java 17 has been recognized by IntelliJ in the **JDK** field, and for the **Java** version, select at least **17**. The version of Spring Boot can’t be found in the dialog, but it can be selected on the next dialog page. Click the **Next** button to get to the dependencies in the next dialog (see Figure 1.8).



New Project

Spring Boot: 3.1.3

☑ Download pre-built shared indexes for JDK and Maven libraries

Dependencies:

Search
Developer Tools
☐ GraalVM Native Support
☐ Spring Boot DevTools
☐ Lombok
☐ Spring Configuration Processor
☐ Docker Compose Support
☐ Spring Modulith
Web
Template Engines
Security
SQL
NoSQL
Messaging
I/O
Ops
Observability

GraalVM Native Support

Support for compiling Spring applications to native executables using the GraalVM native-image compiler.

Added dependencies:

No dependencies added

?
Previous
Create
Cancel

Figure 1.8 Entering Dependencies

The shown *dependencies* are especially supported by Spring. For our first project, we won't need this yet, and we'll add our dependencies later via the POM file as well.

Clicking **Create** completes the process. Now a Maven project has been created, which IntelliJ also starts directly. We can go to the main program under `Date4uApplication` and start it in the IDE. In under two seconds, the application is started and then automatically ends again; that is, after calling the `run(…)` method, which starts the Spring container, the Spring container will end automatically.

IntelliJ IDEA Community Edition

IntelliJ IDEA Community Edition doesn't have framework support out of the box, neither for Spring nor for Jakarta EE. One option is to download a ZIP archive from the Initializr website, unzip it, and import the directory as a Maven project.

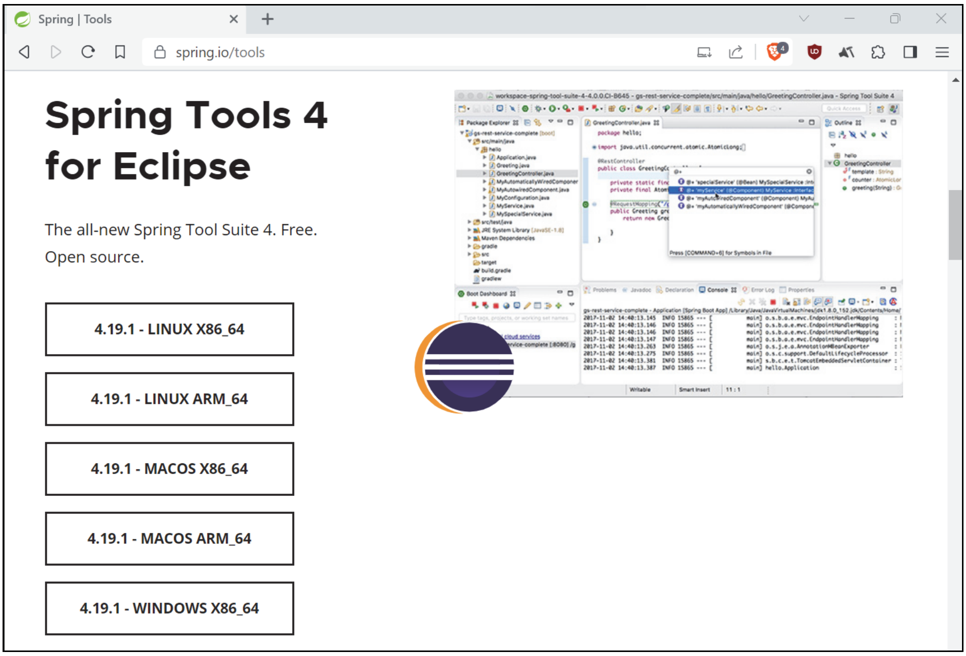
However, there are plug-ins that help us and, like the IntelliJ IDEA Ultimate Edition, replicate a dialog for new projects. These plug-ins are rated and completed differently, and we have no recommendation here. The free extensions don't replace the commercial "big" Ultimate Edition, which detects many configuration errors or can execute SQL queries directly in the Java code.

Spring Tool Suite

As mentioned earlier, VMware has created its own plug-in for the Eclipse IDE: the Spring Tool Suite (STS; *https://spring.io/tools*) (see Figure 1.9). If you want to use the STS, you have two options:

- An installed standard Eclipse is subsequently extended with STS plug-ins via the Marketplace.
- You choose an installation that is based on the current version of the Eclipse IDE and already contains the plug-ins. Periodically, when there is a new version of the Eclipse IDE, there are also updated versions of the STS.

The second option is usually less problematic because it avoids plug-in conflicts.



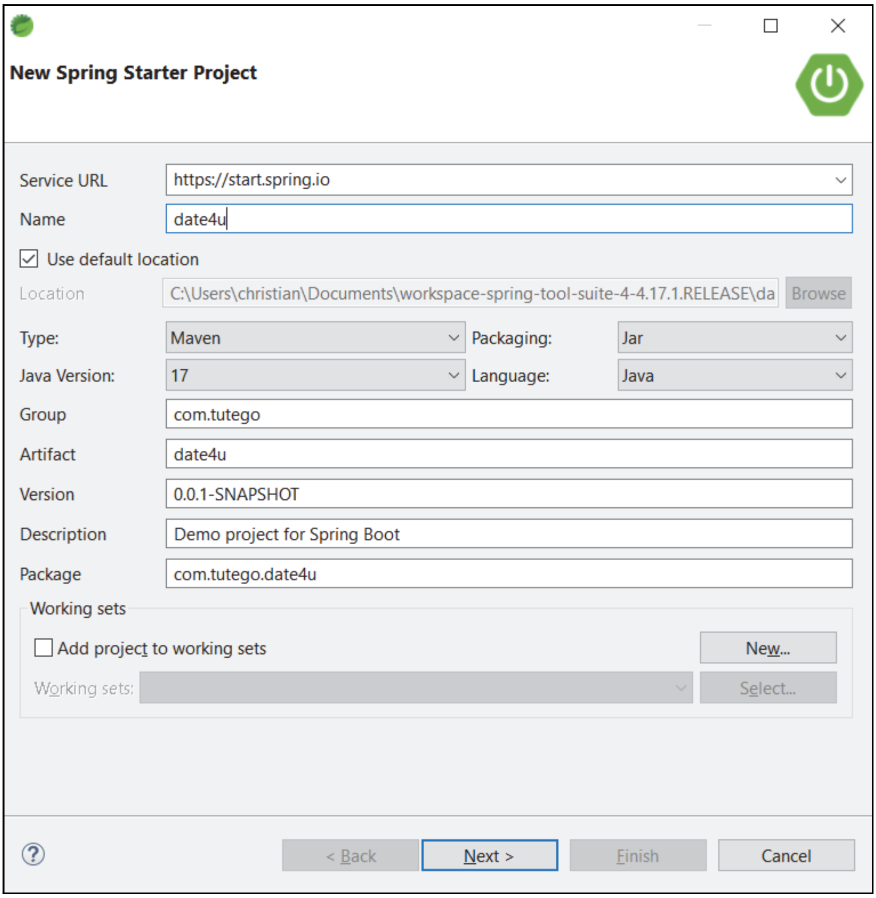
Spring Tools 4 for Eclipse

The all-new Spring Tool Suite 4. Free. Open source.

Figure 1.9 The Eclipse STS Start Page with Download Links

The download file is a *self-extracting JAR* that only needs to be started. After unpacking, there is an EXE file under Windows, and Eclipse can be started. After a short delay, the *Tools Launcher* appears. With Eclipse, there is always a *workspace*, a place for central configurations, where projects are also stored. After the dialog confirmation, Eclipse boots up, and on the left side, the **Create New Spring Starter Project** item appears due to the special configuration of the Eclipse IDE. If this entry doesn't appear there, we can always build on a Spring project under **File • New • Spring Starter Project**.

You can see from Figure 1.10 that the dialog looks a bit similar to what we saw on the Initializr web page. This web service is actually also used as the endpoint because it then generates the ZIP file that Eclipse unpacks and uses as the basis for the project.



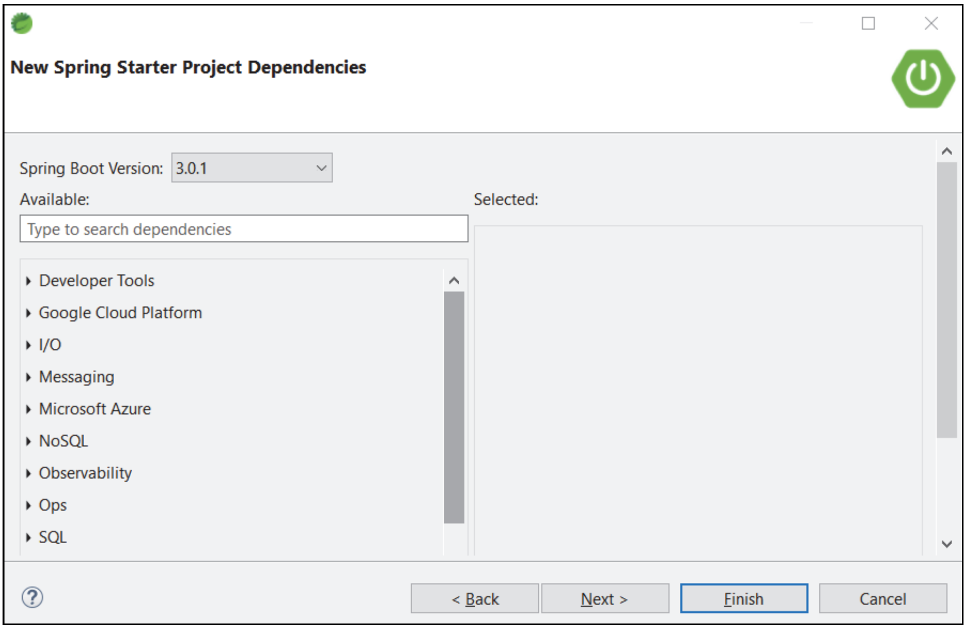
New Spring Starter Project

Service URL: https://start.spring.io
Name: date4u
☑ Use default location
Location: C:\Users\christian\Documents\workspace-spring-tool-suite-4-4.17.1.RELEASE\da
Type: Maven
Packaging: Jar
Java Version: 17
Language: Java
Group: com.tutego
Artifact: date4u
Version: 0.0.1-SNAPSHOT
Description: Demo project for Spring Boot
Package: com.tutego.date4u
Working sets
☐ Add project to working sets

Figure 1.10 Creating a New Spring Boot Project in STS

In the name field, call the project "date4u". The default location is the workspace folder. By default, the Initilizr selects **Gradle** for **Type**, but let's change that to **Maven**. All Java versions starting from Java 17 are allowed; Java **17** is a good choice in the **Java Version** field and is preselected. Enter "com.tutego" as the **Group** and "date4u" for **Artifact**. The **Version** number and the **Description** both fit. For the **Package**, enter "com.tutego.date4u". These settings have nothing to do with Spring; these are the typical settings we would have to make for every Maven or Gradle project.

Click on **Next** to get to the dialog shown in Figure 1.11. This is where it gets interesting because we can select the dependencies that are typical for Spring projects.



New Spring Starter Project Dependencies
Available:
Selected:
Type to search dependencies
Developer Tools
Google Cloud Platform
I/O
Messaging
Microsoft Azure
NoSQL
Observability
Ops
SQL
< Back
Next >
Finish
Cancel

Spring Boot Version: 3.0.1

Figure 1.11 Enter Dependencies

For example, if we wanted to do something with a web server, we could enter "web" in the search box; this narrows down the search. However, we don't need any dependencies for our first projects, so we can skip this part. Click **Finish** to obtain the ZIP archive, and the project is built.

If you navigate to the main class `Date4uApplication`, you can run the project. In the context menu, there are two options under **Run As**: **Java Application** and **Spring Boot App**. The latter option leads to colored output.

Tip

Regarding color, there is a serious problem to consider: Eclipse (at least in the current version) has great difficulty getting colored outputs onto the screen in a performant manner. In other words, if you notice sluggish output and scrolling takes a lot of time, you should turn off color display by deactivating the **ANSI Console Output** item in the configuration.

1.3 Spring Boot Project: Dependencies and Starter

Now, let's delve into the dependencies of a typical Spring Boot project.

1.3.1 Project Object Model with Parent Project Object Model

The structure of the POM file generated by Initializr looks like this:

```
<?xml …?>
<project …>
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    …
  </parent>

  <groupId>com.tutego</groupId>
  <artifactId>date4u</artifactId>
  <version>0.0.1-SNAPSHOT</version>

  <properties>
    <java.version>17</java.version>
  </properties>

  <dependencies>…</dependencies>

  <build>…</build>
</project>
```

Listing 1.1 pom.xml Structure

Using a Parent POM

The default Maven project created by Initializr includes a reference to a parent POM, along with the Maven coordinates such as group ID, artifact ID, and version number. Additionally, the project includes properties for the desired Java version. The dependencies section lists all the required dependencies for the project, while the Maven plug-in section specifies the plug-in used for building and packaging the project.

A parent POM is a kind of "superclass" for Maven projects. It can be used to transfer information from the parent POM to your own project. Spring refers to `org.springframework.boot:spring-boot-starter-parent`:

```
<?xml …?>
<project …>
  …
  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>3.1.3</version>
  </parent>
  …
</project>
```

Listing 1.2 pom.xml

The parent POM defines some properties. These include `java.version` for the Java version number, which Spring Boot 3 sets to Java 17 by default, and `java.version` to initialize `maven.compiler.source` and `maven.compiler.target`. UTF-8 encodings are also set by default, which is common practice, and licenses and some developer information follow as well.

Parent POM also References a Parent POM with a Bill of Materials

Interestingly, the *Spring Boot Starter parent* itself references a parent POM: `org.springframework.boot:spring-boot-dependencies`. This POM defines a *bill of materials* (BOM). Primarily, version numbers of dependencies are declared in the BOM.

When declaring the BOM, the Spring team uses a two-step approach. In the first step, numerous properties are defined. Here is a sample of the POM file from `org.springframework.boot:spring-boot-dependencies`:[6]

```
<properties>
  <activemq.version>5.18.1</activemq.version>
  <angus-mail.version>1.1.0</angus-mail.version>
  <artemis.version>2.28.0</artemis.version>
  <aspectj.version>1.9.19</aspectj.version>
  <assertj.version>3.24.2</assertj.version>
  …
</properties>
```

The keys are usually composed of the artifact ID and the version number. This determines many version numbers of various projects, which are often used in Spring projects. Of course, not all Java projects in the world appear there, but at least the projects that play an important role in the Spring universe do.

The Spring team makes sure that these version numbers of the referenced projects match perfectly. Who wants to do that themselves and check that, for example, the logging library with version number 1.2.3 matches perfectly with the web server of version 3.4.5? Simply setting all version numbers to the latest version doesn't always work. If you raise the version number for one library on one side, this higher version number might become a problem for another library. It's a cumbersome task to ensure that every version perfectly matches; otherwise, there may be problems when running with any outdated Java archives because variables, methods, or types may be missing.

---

[6] *https://repo1.maven.org/maven2/org/springframework/boot/spring-boot-dependencies/3.1.3/spring-boot-dependencies-3.1.3.pom*

After this declaration of the variables, a dependency management block comes later in the POM:

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.apache.activemq</groupId>
      <artifactId>activemq-amqp</artifactId>
      <version>${activemq.version}</version>
    </dependency>
    …
</dependencyManagement>
```

Dependency management means that the named dependencies aren't included in the classpath, but only declared with their version number. When setting the versions, the `<dependency>` block refers to the previously declared properties.

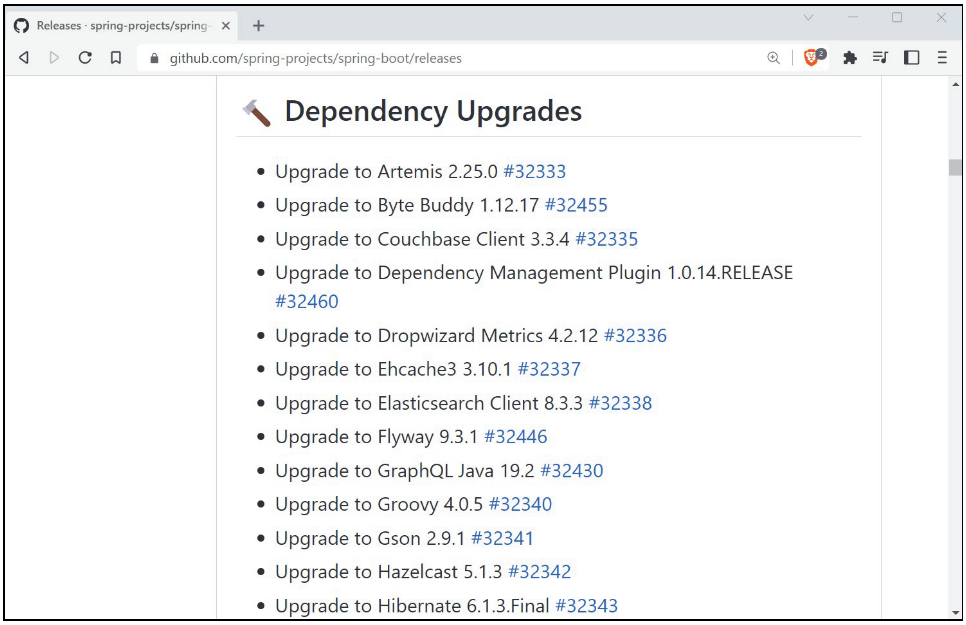
In the POM file of a Spring Boot project, we don't need to specify the version number of a dependency when we add it later. Therefore, the number of version numbers in our POM file is relatively small. While the versions of the referenced Java projects that aren't part of Spring Boot still need to be specified, we don't need to provide version numbers for any Spring-related projects, as they are automatically set to the appropriate version as we've seen.

However, this also means that we must not forget to regularly adjust our Spring Boot version because this is the only way to get the version of the referenced projects readjusted as well. This is an important feature, and if you look at the release notes of the corresponding Spring Boot versions at *https://github.com/spring-projects/spring-boot/releases*, the dependency upgrades always show up (see Figure 1.12).

It's possible to correct the version numbers of individual projects. For this purpose, the predefined variable can be overwritten in the properties block of our POM file. Let's take H2 as an example:

```
<properties>
  <h2.version>1.4.200</h2.version>
</properties>
```

The block sets the predefined property `h2.version` to the desired version. When a dependency on H2 is subsequently added to the POM, Maven takes the set version number. Whether it's higher or lower isn't relevant for Maven.



Releases · spring-projects/spring-
github.com/spring-projects/spring-boot/releases

🔨 Dependency Upgrades

- Upgrade to Artemis 2.25.0 #32333
- Upgrade to Byte Buddy 1.12.17 #32455
- Upgrade to Couchbase Client 3.3.4 #32335
- Upgrade to Dependency Management Plugin 1.0.14.RELEASE #32460
- Upgrade to Dropwizard Metrics 4.2.12 #32336
- Upgrade to Ehcache3 3.10.1 #32337
- Upgrade to Elasticsearch Client 8.3.3 #32338
- Upgrade to Flyway 9.3.1 #32446
- Upgrade to GraphQL Java 19.2 #32430
- Upgrade to Groovy 4.0.5 #32340
- Upgrade to Gson 2.9.1 #32341
- Upgrade to Hazelcast 5.1.3 #32342
- Upgrade to Hibernate 6.1.3.Final #32343

Figure 1.12 Dependency Upgrades of Spring Boot Versions in the Release Notes

1.3.2 Dependencies as Imports

It’s not always necessary to reference Spring Boot dependencies through a parent POM; they can also be imported. This is particularly useful when using a custom parent POM. Some companies define their own parent POM to store information such as version numbers or licenses.

If you want to create a project with Initializr but use your own parent POM, you can replace the default parent POM with your own. However, this can cause issues with missing dependencies, which are critical. To address this, you can set a custom dependency management block with an import dependency. Here is an example:

```xml
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-dependencies</artifactId>
      <version>x.y.z</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
```

Note

There is an important difference between the Spring parent POM and a Spring POM import. Children "inherit" properties, such as UTF-8 encoding, from the Spring Boot Starter parent. These properties must be added manually or defined in a separate parent POM.

1.3.3 Milestones and the Snapshots Repository

The Maven Central server only hosts releases of libraries and not development versions such as snapshots or milestones. However, in the Spring environment, there is a separate server maintained by the Spring team that provides access to snapshots and milestones. This can be useful when working with features that are still under development or when bugs need to be fixed before the next official release. In such cases, it may be necessary to use a daily updated version, even if it's not yet a release.

For milestones and snapshots, copy the following in the POM:

```
<repositories>
  <repository>
    <id>spring-milestones</id>
    <name>Spring Milestones</name>
    <url>https://repo.spring.io/milestone</url>
    <snapshots><enabled>false</enabled></snapshots>
  </repository>
  <repository>
    <id>spring-snapshots</id>
    <name>Spring Snapshots</name>
    <url>https://repo.spring.io/snapshot</url>
    <releases><enabled>false</enabled></releases>
  </repository>
</repositories>
```

Tip

The Spring Initializr site (*https://start.spring.io/*) also allows the creation of Spring boot projects with a milestone or snapshot version. If you select a snapshot and click **Explore**, you can copy the fragment from the displayed POM into your own POM file.

1.3.4 Configuring Annotation Processors

The Spring Initializr builds a POM file that we use to extend various elements, typically by adding additional dependencies. Moreover, the compiler plugin is often configured with annotation processors. An *annotation processor* can process annotations in the source code, generate additional code, or perform certain actions based on the annotations. In the Spring environment, annotation processors are used, for example, to generate JSON files for describing external configurations or meta-model classes.

The integration of an annotation processor can be done in different ways, such as through the `<annotationProcessorPaths>` configuration:

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <configuration>
        <annotationProcessorPaths>
        <! -- -->
        </annotationProcessorPaths>
      </configuration>
    </plugin>
</build>
```

Once we integrate an annotation processor, we will return to the configuration.

1.3.5 Starter: Dependencies of the Spring Initializr

In the POM file, the Initializr has inserted two dependencies:

```
<?xml version="1.0" encoding="UTF-8"?>
<project …>
  …
  <dependencies>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter</artifactId>
    </dependency>

    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-test</artifactId>
      <scope>test</scope>
    </dependency>
  </dependencies>
  …
</project>
```

There is one dependency on the `spring-boot-starter` and a second dependency on `spring-boot-starter-test`, where the second dependency is in scope `test`.

The function of a *starter* is that we don't have to worry about fine-grained dependencies, but we get a whole bundle, a collection of dependencies, via this starter. The official starters all start with a prefix, namely `spring-boot-starter-`, and the group ID is always `org.springframework.boot`. The smallest starter is called *core-starter* and brings everything you need for Spring Boot applications.

Figure 1.13 shows the dependencies: `spring-boot-starter` is on top and has a whole set of dependencies. In other words, if we put a dependency on `spring-boot-starter`, we get what is shown referenced as well. We no longer have to independently pick up, for example, a logging library or the Spring Framework itself because it's all referenced from that `spring-boot-starter`.

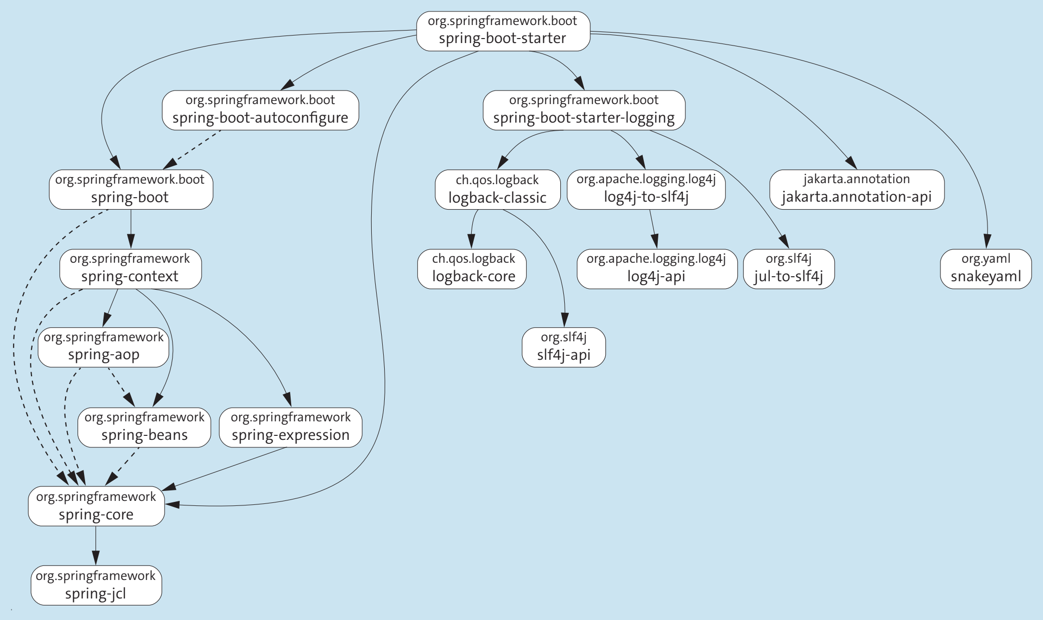

org.springframework.boot
spring-boot-starter
org.springframework.boot
spring-boot-autoconfigure
org.springframework.boot
spring-boot-starter-logging
org.springframework.boot
spring-boot
ch.qos.logback
logback-classic
org.apache.logging.log4j
log4j-to-slf4j
jakarta.annotation
jakarta.annotation-api
org.springframework
spring-context
ch.qos.logback
logback-core
org.apache.logging.log4j
log4j-api
org.slf4j
jul-to-slf4j
org.yaml
snakeyaml
org.springframework
spring-aop
org.slf4j
slf4j-api
org.springframework
spring-beans
org.springframework
spring-expression
org.springframework
spring-core
org.springframework
spring-jcl

Figure 1.13 Dependencies of the Core Starter

On the left side, we can see that `spring-context` references the Spring Framework itself, including its dependencies. On the right side we see the dependency to `spring-boot-starter-logging`, that is, to the logging infrastructure. Then, a few standard annotations are included, and the YAML Ain't Markup Language (YAML) parser *SnakeYAML* is also included.

More Spring Boot Application Starters

Numerous other starters are also available. The following list shows only a selection:

- `spring-boot-starter-jdbc` for applications with database accesses via JDBC and DataSources
- `spring-boot-starter-data-jpa` for applications with Jakarta Persistence

- `spring-boot-starter-json` for the JavaScript Object Notation (JSON) mapping
- `spring-boot-starter-web` for web services and dynamic websites, including the servlet container Tomcat

Tip

An overview of all starters is provided at *https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#using-boot-starter*.

With all starters, a specific technology can be easily integrated without us having to worry about the little details and a bunch of sub-dependencies when we, for example, want to access a relational database, develop a web service, work with the WebSocket API, or do something with security.

The starters aren't mutually exclusive, so they can be used in combination. If you want to build a web application with a database connection, you can use a starter for the web together with a starter for Jakarta Persistence, for example.

Some starters contain the core starter, so that it's not mandatory to include it. However, you don't have to remember this because including the core starter and web starter, for example, isn't a mistake.

Note

The official starters all *start with* `spring-boot-starter-`, such as `spring-boot-starter-web` or `spring-boot-starter-jdbc`. Starters that don't come from VMware should have a well-defined name and start with the project name, *followed* by `-spring-boot-starter`. Examples from the open-source world include `grpc-spring-boot-starter` or `okta-spring-boot-starter`.

1.4 Getting Started with Configurations and Logging

The Initializr creates various documents, among them the *application.properties* file, which is empty by default. This file is useful for configuring Spring Boot applications. Spring Boot has various property sources that refer to the sources from which Spring Boot pulls configurations. Two of these sources are *application.properties* and *application.yml*; the command line is another.

Usually, the file is located in the classpath, that is, in the source code folder *src/main/resources*. Everything in this folder will later be in the root directory of the built application.

1.4.1 Turning Off the Banner

The first thing we need to do is turn off the banner that appears at startup:

```
spring.main.banner-mode=off
```

Listing 1.3 application.properties

If you start the program again, the banner disappears.

There are three valid assignments for `spring.main.banner-mode`:

- `off`
  Switch off completely.
- `console`
  This is the default; the banner appears on `System.out`.
- `log`
  Write the banner to the current log stream.

IntelliJ Tip

A good development environment "knows" almost all configuration properties and valid values, and keyboard completion should be possible. If you enter "spring.main.banner-mode=", then the development environment knows the three possible configuration assignments: `off`, `console`, and `log`. The documentation is also displayed.

1.4.2 Logging API and Simple Logging Facade for Java

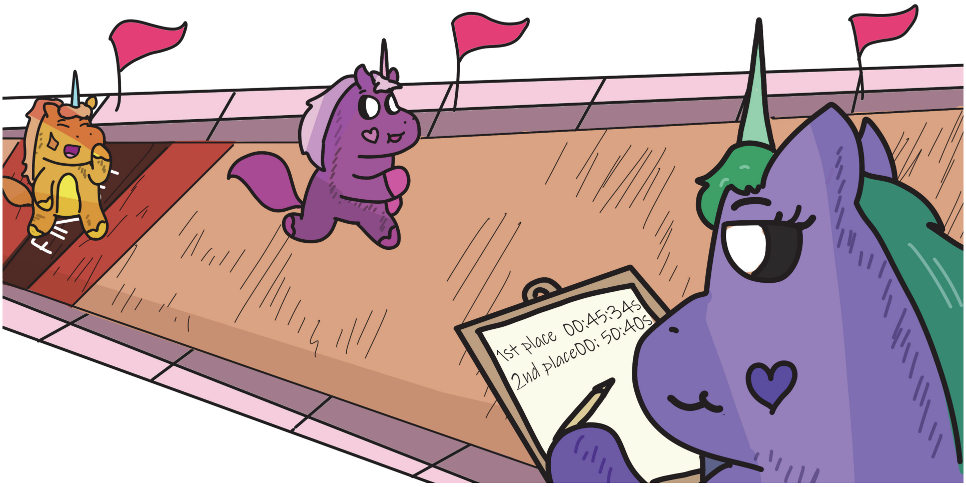
The `spring-boot-starter` has a dependency on `spring-boot-starter-logging`. This starter provides a logging infrastructure and configures two logging facades:

- *Simple Logging Facade for Java* (SLF4J) (*www.slf4j.org*)
- *Commons Logging API* (*https://commons.apache.org/proper/commons-logging*)

Note

The Spring Framework uses the Commons Logging API facade internally; however, they have actually wanted to get away from it for years.

Typically, it's uncommon to work directly with a logging library such as Logback or Log4j. Instead, *logging facades* are often used. A *facade* is a well-known design pattern that accepts requests from external sources and then delegates them to a complicated subsystem. By logging solely through a facade, the details of the underlying logging libraries are concealed. This enables logging implementations to be modified with ease. For instance, today you might use Logback, but tomorrow Log4j 2 may be preferred.



Using the SLF4J API

Now let's program a small example with the SLF4J library, which requires only three steps:

1. Set an `import` declaration for the types:

   ```
   import org.slf4j.*;
   ```

2. Declare a variable `log`:

   ```
   private final Logger log = LoggerFactory.getLogger( getClass() );
   ```

   Whether the variable is `static` or not depends a bit on the use case.

3. Use the appropriate logging methods, for example, placed in the `main(…)` method:

   ```
   log.info( "Log with arguments {}, {} and {}", 1, "2", 3.0 );
   ```

Logger methods are called `debug(…)`, `error(…)`, `info(…)`, and so on. They indicate the urgency with which something should be reported. With the `log.info(…)` method call, you'll see something comparable to `System.out.printf(…)`, which is a format string, but with the curly braces as placeholders, it's much simpler than a Java `Formatter`. With the placeholders, it's all about order. If it's `log.info("Log with arguments {}, {} and {}", 1, "2", 3.0)`, it will end up with `Log with arguments 1, 2, and 3.0`.

Log Level

Different log methods are offered by the `Logger` object, for example, `debug(…)`, `info(…)`, `error(…)`, but it's not self-evident that all outputs appear. Let's consider the following:

```
log.debug( "Debug Level Log" );
log.info( "Info Level Log" );
log.error( "Log with arguments {}, {} und {}", 1, "2", 3.0 );
```

All three outputs won't appear on the console because by default the debug message doesn't appear on the console. This is caused by the *log level*. The log level defines an urgency, which must be at least given, so that a message is written. SLF4J defines the log levels `TRACE`, `DEBUG`, `INFO`, `WARN`, `ERROR`, `FATAL`, and `OFF` (for switching off messages). If the `debug(…)` messages don't appear, this is because the log level is currently different; that is, the urgency is above `DEBUG`.

The default log level is set to `INFO`, which indicates that only messages with an urgency level of `INFO` or higher, such as `WARN`, `ERROR`, and `FATAL`, will be written to the console. Messages with a lower urgency level, such as `DEBUG` and `TRACE`, won't be logged at the `INFO` log level.

Setting the Log Level

The log levels can be set via configuration properties. The following could be set in the *application.properties* file:

```
logging.level.com.tutego.date4u=DEBUG
logging.level.org.springframework=ERROR
```

The configuration properties start with the prefix `logging.level`. This is followed by a package specification or a fully qualified type. In our example, we set the log level of `com.tutego.date4u` to `DEBUG`. The log level is not only set for this package but also for all subpackages. For example, if there is a class with a `debug(…)` message under `com.tutego.date4u.very.deep.nested`, the log message will also appear.

`logging.level` can also be followed by a concrete type. This is useful if, for example, the log level is to be set for concrete classes, but not for all other types in the same package at the same time. A specific log level for a type or package overrides a parent setting.

The logging output of the Spring framework can also be configured in this way. For example, if the `logging.level` of `springframework` is set to `ERROR`, anything less important than `ERROR` won't be output. If you want to see everything that the Spring Framework logs, set the log level of `org.springframework` to `TRACE`. However, the massive output of these logs slows down the application.

[+]

Tip

On production systems, output (e.g., log messages) via the console is rare because it's slow; however, the situation is different when log streams are redirected to files. On top of that, a program should only log what is evaluated later. Care should be taken that no security-relevant data is logged and stored. For Java applications in containers (Docker, Kubernetes cluster), log outputs are written to the console.

1.5 Summary

By way of introduction, this chapter showed the difference between the Java SE platform, Jakarta EE, and Spring. Spring uses many Jakarta EE standards, but it isn't a compatible Jakarta Enterprise application server.

Next, we saw how the Spring Initializr—via the web page or built into the development environment—helps to build a new Spring Boot project. You can also do this via the *Spring Boot CLI*, a command-line tool, but this is rarely used in practice, so I refer you to the reference documentation at *https://docs.spring.io/spring-boot/docs/current/reference/html/cli.html*.

After we've built the project, we can deal more intensively with the Spring container in the next step. The Spring container manages Spring-managed beans. We need to learn how to add new components to the Spring container and request them.

Chapter 2

Containers for Spring-Managed Beans

In this chapter, we'll take a closer look at the Spring container because that is where Spring-managed beans live.

2.1 Spring Container

At the heart of a Spring application is a special data structure, the Spring container (*container* for short). The objects that this container manages are called *Spring-managed beans* or *components*.

The container has three central tasks:

- Create
- Manage the Spring-managed beans
- Configure and resolve dependencies between components

Components can later be taken out of the container and queried.

Not all objects are inside the Spring container, and there are still plenty of objects outside the container, such as strings or loaded rows from a database (see Figure 2.1).

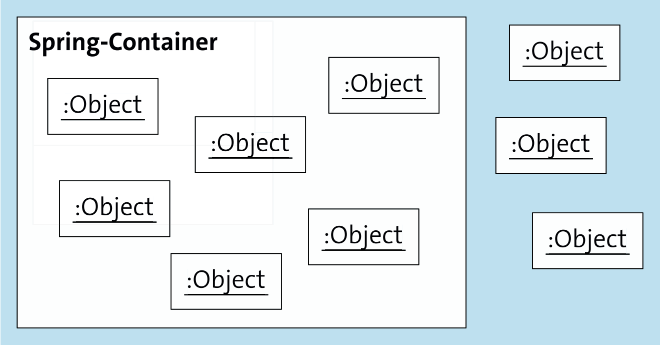


Figure 2.1 Spring Container with Spring-Managed Beans

2.1.1 Start Container

The *Initializr* has generated a class called `Date4uApplication` with a `main(…)` method. In the body of the `main(…)` method, there is a `run(…)`call, which starts up the container.

```
@SpringBootApplication
public class Date4uApplication {
  public static void main( String[] args ) {
    SpringApplication.run( Date4uApplication.class, args );
  }
}
```

Listing 2.1 Date4uApplication.java

The `run(…)` method is a static method of the `SpringApplication` class.[1] Two pieces of information are passed to the `run(…)` method: the *initial configuration*, which here is the type of your own class, and the actual parameters given to the `main(…)` method, which are passed to this `run(…)` method. Spring takes out certain configuration information from the arguments. If these arguments weren't passed, Spring would not be able to get the arguments we pass on the command line.

Three things can happen when `run(…)` is called:

- If there is an error in the configuration, the container isn't started and the program stops; `run(…)` ends with an exception.
- Everything is fine, the container initializes all components and is then finished, and the `run(…)` method is exited. However, a web server such as Tomcat could have been started, for example, and the program continues to run even though the `main(…)` method ended.
- The `run(…)` method may hang in an infinite loop because, for example, an interactive shell is started.

Besides the static `run(…)` method of the `SpringApplication` class, there are two other ways to start the container. These options are interesting because they allow us to configure the container right at the start.

2.1.2 Instantiate SpringApplication

The first alternative to the static `run(…)` method is to build an object of the `SpringApplication` type and later use the `run(…)` instance method. The advantage of this variant is that we can make some settings in the `SpringApplication` object:

```
SpringApplication app = new SpringApplication( Date4uApplication.class );
app.setHeadless( false );
app.setBannerMode( Banner.Mode.OFF );
app.setLogStartupInfo( false );
// More initializations
app.run( args );
```

---

[1] *https://docs.spring.io/spring-boot/docs/current/api/org/springframework/boot/SpringApplication.html*

The example shows three settings:

- By default, Spring applications run in *headless mode*, which means they have no graphical user interface (GUI) part. If Spring applications are to have a GUI, this would cause an exception in this mode because the AWT system is disabled. If you want to write GUI applications, you have to turn off the headless mode to get a head again (delightful, this double negation).
- The start banner is turned off. We've seen that a configuration property can turn off the banner. This is also possible in code with the `setBannerMode(…)` method. Enumeration constants such as `Banner.Mode.OFF` are passed so that the banner is no longer output at startup.
- Besides the banner, other startup information is output, including the profile. These outputs can be switched off with `setLogStartupInfo(false)`.

If you look at the static `run(…)` method in implementation, it only builds a `SpringApplication` instance. The static `run(…)` method is just a facade—a shortcut for actually instantiating `SpringApplication`.

The container is started with the instance method `run(…)`. The method is passed the command line arguments in the same way as the static method. The type information doesn't have to be passed to the instance method `run(…)` because it has already been provided to the constructor of `SpringApplication`.

2.1.3 SpringApplicationBuilder *

The second variant is the use of class `SpringApplicationBuilder`,[2] whose name hints at what it's all about: building an object using the builder pattern.

> **Note: Builder Pattern**
>
> With the *builder pattern*, no parameterized constructors or setters are used for configuration, but methods are cascaded, that is, set one after the other. If methods of the type `a().b().c()` are set one after the other, this is also referred to as a *Fluent API*.

The builder pattern is popular to get to a final configuration via Fluent API over several steps. What we've just written can be written this way with the builder:

```
new SpringApplicationBuilder( Date4uApplication.class )
                .headless( false )
                .bannerMode( Banner.Mode.OFF )
                .logStartupInfo( false )
                .run( args );
```

---

[2] *https://docs.spring.io/spring-boot/docs/current/api/org/springframework/boot/builder/SpringApplicationBuilder.html*

The code looks much “airier” this way because where setters were used before, with `SpringApplicationBuilder`, these methods can be easily cascaded for setting.

The settings options of `SpringApplication` and the Spring `ApplicationBuilder` are almost identical. There are two small differences:

- With `SpringApplicationBuilder`, you can only set the states; you can’t read them. That is, there are no getters. In `SpringApplication`, there are setters and getters, but the getters are rarely needed anyway.
- In `SpringApplicationBuilder`, hierarchical contexts can be set. In a hierarchical context, parents, children, and siblings can be modeled to have an isolated container, which can guarantee that, for example, there is no accidental access to the children from the parent component. We’ll look at how this works in Section 2.8.9.

2.1.4 What main(…) Does and Doesn’t Do

Not much happens in the `main(…)` method in general, except that `run(…)` starts up the container. The reason is that in the `main(…)` method, we’re looking at this container with the Spring-managed beans from the outside, so to speak. However, the components of our application must themselves be part of the Spring container to be mutually connected.

That’s why it’s important to think about how to run your own program code in the context of the container. There are several possibilities, which we’ll look at in a moment in Chapter 3, Section 3.3.

2.1.5 The run(…) Method Returns ConfigurableApplicationContext

The `run(…)` method returns an *application context* (or *context*, for short). This can be stored in a variable and used later:

```
ConfigurableApplicationContext ctx =
  SpringApplication.run( Date4uApplication.class, args );
```

`ConfigurableApplicationContex`[3] is an interface that extends other interfaces; it’s a rather complex inheritance relationship, as Figure 2.2 shows.

Fortunately, it’s not at all necessary to know all this, but from the type names, you can get the first clues:

- The Spring context is a `BeanFactory` that manages Spring-managed beans at its heart.
- `MessageSource` can include internationalized resource files.
- Resources can be loaded using the `ResourceLoader` (see Chapter 3, Section 3.5.2).
- Events can be sent to and received from the container (see Chapter 3, Section 3.4).

---

[3] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/context/ConfigurableApplicationContext.html*

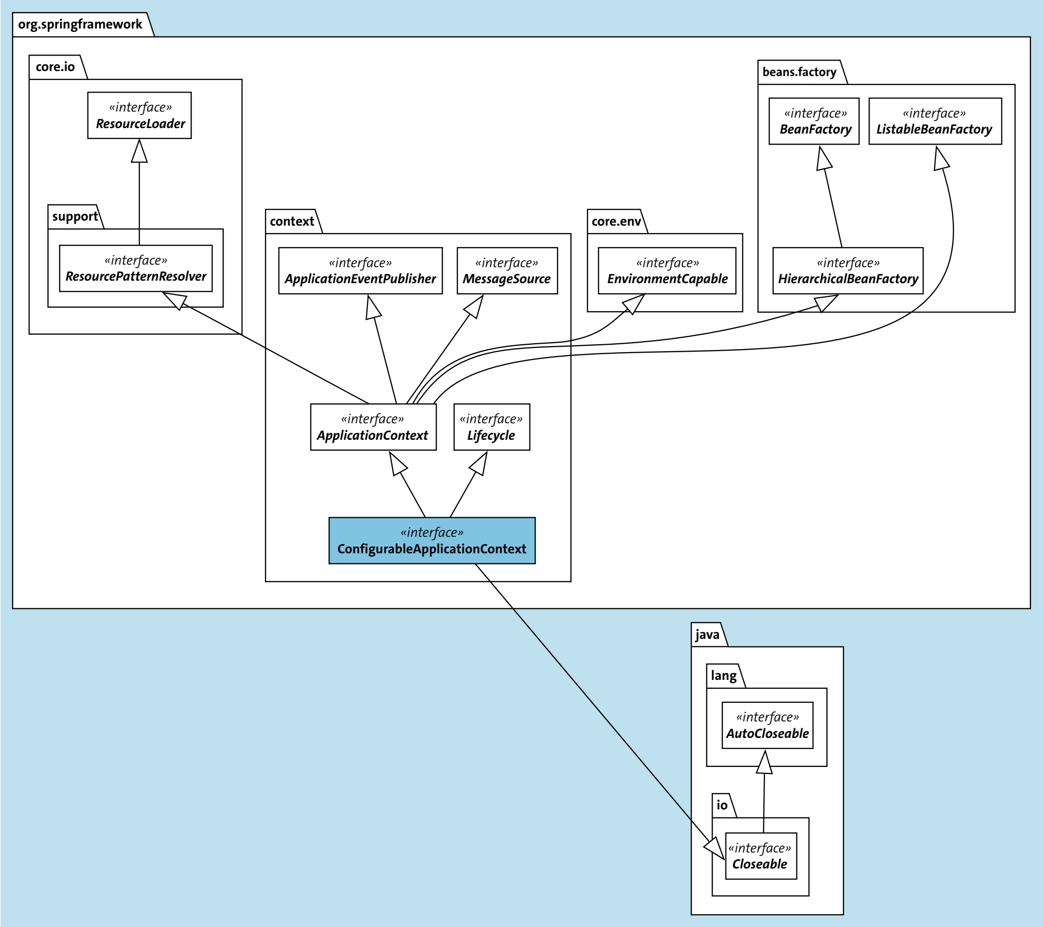

org.springframework
core.io
«interface»
ResourceLoader
support
«interface»
ResourcePatternResolver
context
«interface»
ApplicationEventPublisher
«interface»
MessageSource
core.env
«interface»
EnvironmentCapable
beans.factory
«interface»
BeanFactory
«interface»
ListableBeanFactory
«interface»
HierarchicalBeanFactory
«interface»
ApplicationContext
«interface»
Lifecycle
«interface»
ConfigurableApplicationContext
java
lang
«interface»
AutoCloseable
io
«interface»
Closeable

Figure 2.2 Upper Types of "ConfigurableApplicationContext"

Of course, we'll take a closer look at these points later.

2.1.6 Context Methods

Interface `ApplicationContext` extends `ListableBeanFactory`,[4] which declares method `getBeanDefinitionNames(…)` that returns a `String[]` with the names of the components that the container has registered.

Let's use this method for a sorted listing of all registered components:

---

[4] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/beans/factory/ListableBeanFactory.html*

```
ApplicationContext ctx =
  SpringApplication.run( Date4uApplication.class, args );
Arrays.stream( ctx.getBeanDefinitionNames() )
      .sorted()
      .forEach( System.out::println );
```

The fact that the return of `run(…)` is a `ConfigurableApplicationContext` isn't needed in the example; the type name `ApplicationContext` is shorter and also extends `ListableBeanFactory`.

After starting the modified program, a sorted listing of all logged-in Spring-managed beans appears on the console, which are present without us having configured anything. The output of the approximately 50 Spring-managed beans includes the following:

- `applicationAvailability`
- `applicationTaskExecutor`
- **`date4uApplication`**
- `forceAutoProxyCreatorToUseClassProxying`
- `lifecycleProcessor`
- `mbeanExporter`
- `mbeanServer`
- `objectNamingStrategy`
- `org.springframework.aop.config.internalAutoProxyCreator`
- [ommmited for brevity]
- `spring.task.scheduling-org.springframework.boot.autoconfigure.task.TaskSchedulingProperties`
- `springApplicationAdminRegistrar`
- `sslBundleRegistry`
- `sslPropertiesSslBundleRegistrar`
- `taskExecutorBuilder`
- `taskSchedulerBuilder`

That's actually amazing because we didn't do anything special and still get a number of components put in the container.

Spring Boot Auto-Configuration

This feature is the key difference between Spring Boot and Spring Framework. With Spring Framework, we have to configure everything—nothing happens by itself. With Spring Boot, on the other hand, components are automatically registered because an auto-configuration decides that it's useful to build certain objects. An *auto-configuration*, and the name says it, automatically builds configurations and is the magic wand of Spring Boot—we'll talk about this in great detail for the next hundred pages.

It's nice that there are about 40 components in the context, but currently it doesn't matter for us; these are details we have nothing to do with. Of all the components, one should catch our eye, however: `date4uApplication`. Our own initial configuration was instantiated as an object and then placed in the container. Our component is automatically named; it's lowercase and not capitalized like the class name.

[+]

Tip

Class `BeanFactoryUtils` provides additional methods, for example, to determine all the beans in packages.

Query Targeted Beans with Methods from BeanFactory

Interface `ConfigurableApplicationContext` extends `ListableBeanFactory`. Of type `ListableBeanFactory`, we've already used method `getBeanDefinitionNames()`. `ListableBeanFactory` itself extends `BeanFactory`.[5] There we find methods for taking Spring-managed beans and for existence tests. Here'ss a small method selection:

- `boolean containsBean(String name)`
- `T getBean(Class<T> requiredType)`
- `Object getBean(String name)`
- `T getBean(String name, Class<T> requiredType)`

With `getBean(…)`, a bean name or a type token can be passed, which looks something like this:

```
ApplicationContext ctx =
  SpringApplication.run( Date4uApplication.class, args );
MyBean bean = ctx.getBean( MyBean.class );
```

The return of the methods is the bean or an exception if the bean can't be provided.

All Spring-managed beans have a name (an ID), and also alternative names (called *aliases*) are possible. The beans can be queried by their name. Unlike the `getBean(Class)` method, `getBean(String)` doesn't know what type is returned, and therefore the return type is only `java.lang.Object`. If the type is known, `getBean(String, Class)` returns that return type.

---

[5] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/beans/factory/BeanFactory.html*

2.2 Package Structure of the Date4u Application

The Java classes (except for the main program `Date4uApplication`) are in three subpackages:

- `com`
  - `tutego`
    - `date4u`
      - `core`
      - **`interfaces`**
      - **`infrastructure`**

These subclasses are described here:

- **`core`**
  In the sense of domain-driven design, these are the classes with the business logic. They have no dependencies on types in `infrastructure` and `interfaces`. In the onion architecture, this forms the core. All dependencies are from the outside to the inside.
- **`interfaces`**
  This package contains code that allows data to be requested from outside and functionality to be triggered. Often, this is understood to be a *graphical user interface*, but *interface* is more general. In our case, the classes for the shell and for implementing a RESTful application programming interface (API) are in the subpackage. The classes use functionalities from `core`.
- **`infrastructure`**
  Currently this package is empty, but Spring-specific configurations could be put in here.

In the listings of the Java projects, the packages aren't explicitly listed. The directories and package structure of the finished Date4u project looks like this:

```
+---src
|   +---main
|   |   +---java
|   |   |   \---com
|   |   |       \---tutego
|   |   |           \---date4u
|   |   |               |   Date4uApplication.java
|   |   |               +---core
|   |   |               |   |   FileSystem.java
|   |   |               |   |   Statistic.java
|   |   |               |   +---configuration
|   |   |               |   |       Date4uProperties.java
|   |   |               |   +---event
|   |   |               |   |       NewPhotoEvent.java
|   |   |               |   +---photo
|   |   |               |   |       AwtBicubicThumbnail.java
|   |   |               |   |       AwtNearestNeighborThumbnail.java
|   |   |               |   |       Photo.java
|   |   |               |   |       PhotoService.java
|   |   |               |   |       Thumbnail.java
|   |   |               |   \---profile
|   |   |               |           Profile.java
|   |   |               |           ProfileRepository.java
|   |   |               |           Unicorn.java
|   |   |               |           UnicornRepository.java
|   |   |               +---infrastructure
|   |   |               \---interfaces
|   |   |                   +---rest
|   |   |                   |       LastSeenStatistics.java
|   |   |                   |       PhotoRestController.java
|   |   |                   |       ProfileDto.java
|   |   |                   |       ProfileRestController.java
|   |   |                   |       QuoteRestController.java
|   |   |                   |       StatisticRestController.java
|   |   |                   \---shell
|   |   |                           FsCommands.java
|   |   |                           PhotoCommands.java
|   |   |                           RepositoryCommands.java
|   |   \---resources
|   |       |   application.properties
|   |       |   tutego.date4u.p12
|   |       +---db
|   |       |   \---migration
|   |       |           V1_0__Create_all_tables.sql
|   |       |           V1_1__Add_foreign_keys_references.sql
|   |       |           V2_0__Insert_initial_data.sql
|   |       \---static
|   |               chart.html
|   |               index.html
|   \---test
|       +---java
|       |   \---com
|       |       \---tutego
|       |           \---date4u
|       |               |   Date4uApplicationTests.java
|       |               +---core
|       |               |   |   FileSystemTest.java
|       |               |   +---photo
|       |               |   |       PhotoServiceTest.java
|       |               |   \---profile
|       |               |           ProfileRepositoryTest.java
|       \---resources
|               test.properties
```

2.3 Pick Up Spring-Managed Beans through Classpath Scanning

This section is about how we can add new beans to the container. We've seen that the container already contains a large number of Spring-managed beans, and we'll look at why this is the case. But the central question is: How can we get our own components into the container?

2.3.1 Fill Container with Beans

Basically, there are two different approaches: declaratively or by registering components manually. *Declarative* means that the container recognizes and registers components on its own via a "magic" mechanism. There are two different ways to do this. One is the modern approach of using annotations to declaratively mark components so that they can become a Spring-managed bean. The other way is somewhat older. Here the components can be described via XML files, so that Spring can register the components after reading the XML file.

The modern approach is shown next with an example.

Abstraction of a File System: FileSystem

The first thing we want to do is develop a component for an abstraction of a file system because in our Date4u application, it should of course be possible to store photos. The photos could be stored in a database, but a relational database is less suitable for this. There are multimedia databases that can handle large amounts of data, but we simply want to use the file system.

To abstract from directories, we want to implement a `FileSystem` class for a logical file system:

```
public class FileSystem {
  private final Path root =
    Paths.get( System.getProperty("user.home") ).resolve( "fs" );

  public FileSystem() {
    try { if ( ! Files.isDirectory(root) ) Files.createDirectory(root); }
    catch ( IOException e ) { throw new UncheckedIOException( e ); }
  }
  public long getFreeDiskSpace() {
    return root.toFile().getFreeSpace();
  }

  public byte[] load( String filename ) {
    try { return Files.readAllBytes( root.resolve( filename ) ); }
    catch ( IOException e ) { throw new UncheckedIOException( e ); }
  }

  public void store( String filename, byte[] bytes ) {
    try { Files.write( root.resolve( filename ), bytes ); }
    catch ( IOException e ) { throw new UncheckedIOException( e ); }
  }
}
```

Listing 2.2 FileSystem.java

This logical file system uses a part of the physical file system, namely a subdirectory *fs* in the user directory. The root directory of the logical file system is available in the `root` instance variable for the method calls. All resources are stored in this *fs* directory. The constructor checks if this directory exists and, if not, creates it.

The following methods are used:

- `getFreeDiskSpace()` returns the free bytes. Because `Path` doesn't have any methods to request the storage space, the query runs via class `java.io.File`.
- `load(…)` loads a file. The path is composed of the `root` and the passed file name. The checked exception is caught and wrapped in an `UncheckedIOException`.
- `store(…)` stores a byte array and also resolves the path relatively. The exception handling is the same as for `load(…)`.

2.3.2 @Component

The `FileSystem` class is a regular class with a parameterless constructor. We could easily instantiate this class with `new`. However, we no longer want to instantiate this class ourselves, but it should be created and registered by the Spring Framework. There are several ways to do this: declaratively via XML, declaratively via annotations, or programmatically with code.

We want to take the approach of using an annotation to tell the framework that the class is a component and should be automatically recognized. The most general annotation is called `@Component`.[6] We write the following:

---

[6] *http://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/stereotype/Component.html*

```
@Component
public class FileSystem { … }
```

Listing 2.3 FileSystem.java Extension

The annotation comes from the `org.springframework.stereotype` package, and it's clear: this is an annotation from Spring Framework and has nothing to do with Spring Boot. The annotation type header looks like this:

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Indexed
public @interface Component {
  // …
}
```

The annotation type `@Component` is itself annotated, which is called *meta-annotations*.

The `@Target(ElementType.TYPE)` value indicates that `@Component` may only be attached to type declarations. `@Retention` says that the annotation is accessible at runtime via reflection. The use of `@Documented` expresses that a placed `@Component` annotation itself appears in the Java documentation of that class, and `@Indexed` is a special notation from the Spring context.

If we start the Date4u application and ask for a list of all components, the file system would show up under the name “filesystem”.

In summary, all classes annotated with `@Component` are automatically detected, instantiated, and come into the container's belly as a Spring-managed bean. One reason for this is the annotation `@SpringBootApplication`.

SpringBootApplication Annotation

Let's recall the first program:

```
@SpringBootApplication
public class Date4uApplication {
  public static void main( String[] args ) {
    SpringApplication.run( Date4uApplication.class, args );
  }
}
```

The class was annotated with `@SpringBootApplication`.[7] Let's take a closer look at this annotation. The declaration is as follows:

---

[7] *https://docs.spring.io/spring-boot/docs/current/api/org/springframework/boot/autoconfigure/SpringBootApplication.html*

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@SpringBootConfiguration
@EnableAutoConfiguration
@ComponentScan(excludeFilters = {
   @Filter(type = FilterType.CUSTOM,
                  classes = TypeExcludeFilter.class),
   @Filter(type = FilterType.CUSTOM,
                  classes = AutoConfigurationExcludeFilter.class) })
public @interface SpringBootApplication {
  …
}
```

Annotation `@SpringBootApplication` is a compound annotation type. For example, it's annotated `@ComponentScan`; we'll discuss the filters later in Section 2.3.4. Annotation `@ComponentScan` gives the Spring Framework a clue that it should automatically search for annotated types when the container starts. If `@ComponentScan` is missing, the components won't be recognized.

Root of Classpath Scanning

If the container is started with `run(…)`, the Spring Framework searches for `@Component` annotated types. Now you need to know different things, for example, that it doesn't search for annotated classes everywhere—that would be a bit much. There is a restriction on where the container starts searching. By default, the container starts at the package of the main configuration, which, in our case, is the class with the `main(…)` method.

Let's assume the following package structure:

- com
  - tutego
    - date4u
      - Date4uApplication.class
    - core
      - FileSystem.class
      - DifferentClass.class
    - package2
      - AnotherClass.class

If the `main(…)` method is in a class in the `com.tutego.date4u` package, then, by default, the container searches in that package, including all subpackages.

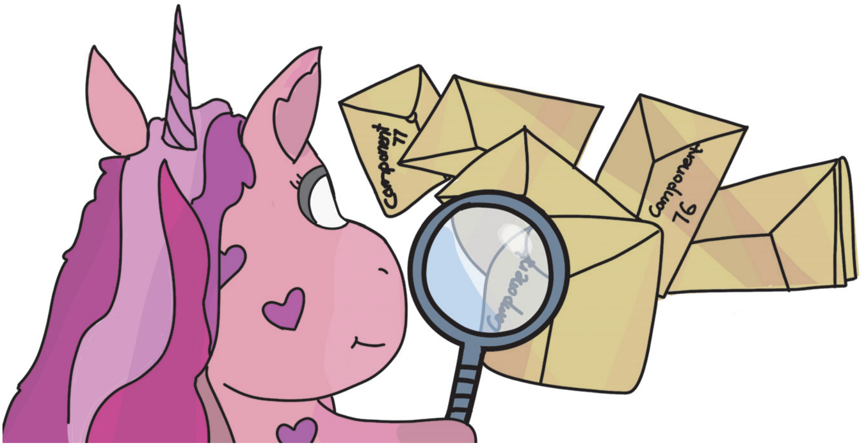
If these types are in completely different packages, such as `de.javatutor`, then these classes would not be recognized by default. Classpaths can be added, of course, and this is important—especially when we mix software from other teams that have a different package structure. Then these packages must be explicitly included; we'll discuss how to do this in Section 2.3.4.

The reason not all classes from all packages are searched is, among other things, the performance. Opening the appropriate class files and looking for whether an annotation is set comes with computing time costs, which we want to avoid. Therefore, the search is limited to the classes that lie in our own package and in the subpackages.

Note

Types should never be in the default package, which is the unnamed package. This is bad style to be avoided.

Let's briefly repeat the process of classpath scanning: The moment the container starts with `run(…)`, it will know via `@ComponentScan` that it should start classpath scanning. It will get a list of all classes, open it and look inside. If those classes are annotated with `@Component` (there are other annotations, more on that in a moment), then the container will automatically instantiate that class. A Spring-managed bean is created, put into the container, and appears consequently with `getBeanDefinitionNames()`.



2.3.3 @Repository, @Service, @Controller

Besides the `@Component` annotation, there are other annotations that lead to a new Spring-managed bean. We want to get to know `@Repository`, `@Service`, and `@Controller`. These annotations are called *semantic annotations*. Here, the term *stereotype* actually fits a bit better. A generic `@Component` just says it's "some component." But components have a purpose. To better document them, there are three annotations:

- `@Service` classes normally execute the business logic.
- `@Repository` classes usually go to data stores.
- `@Controller` classes accept requests from the frontend.

This results in a data flow as follows: There is a request from the client, which a controller receives. The controller delegates the tasks to the service. The service needs data, so it goes to the repository; the repository usually delivers the data from a database. The data goes back to the service, and the service delivers the data back to the controller. A controller is a component such as a REST controller for RESTful web services, or a controller for a command-line interface or for a chat interface.

So far, these three special annotations have no special semantics, which means that whether a component is annotated with `@Component` or with `@Repository`, `@Service`, or `@Controller`, basically doesn't matter. But it can be, and so it's written in the reference documentation of the Spring Framework that this will change in the future. So, it wouldn't be good if a program wants to express an `@Service` but is set as `@Repository`.

Let's come back to the `FileSystem` component. The generic annotation `@Component` doesn't really fit; `@Repository` and `@Service` fit better. Let's change it so that `@Component` becomes `@Service`:

```
@Service
public class FileSystem { … }
```

Listing 2.4 FileSystem.java Extension

Our Own Composed Annotations

Let's quickly look at the implementation of the `@Service` annotation type:

```
@Target(value=TYPE) …
@Component
public @interface Service { … }
```

This kind of thing is quite common in the Spring environment: annotation types with meta-annotations. We can do this as well: build our own annotation types, for example, `@FilesystemService`. We would then annotate this annotation type with `@Service`, for example, and the result would be the same: a class annotated with `@FilesystemService` or `@Service` would be recognized during classpath scanning and included as a new Spring-managed bean.

Get FileSystem from a Container via Context

Components can be specifically queried from the container with `getBean(…)`, and so can our `FileSystem`. Let's do this to write the number of free gigabytes to the console:

```
ApplicationContext ctx = SpringApplication.run( Date4uApplication.class, args );

FileSystem fs = ctx.getBean( FileSystem.class );
System.out.println(
  DataSize.ofBytes( fs.getFreeDiskSpace() ).toGigabytes() + " GB"
);
```

To convert to gigabytes, you would probably use division, but the nice thing is that the Spring Framework has its own data type called `DataSize` for data sizes. The static factory method `ofBytes(…)` builds the `DataSize` object with bytes, and `toGigabytes()` returns the output in gigabytes.

2.3.4 Control Classpath Scanning More Precisely with @ComponentScan *

Spring can conveniently find the types itself via classpath scanning. This section deals with the configuration of the search.

Start Configuration

The `run(…)` method is passed a start configuration. The start configuration is either marked with `@Configuration`, or it's an `@Component`. (`@Configuration` is a specialization of `@Component`, a *Java configuration class*.) Instead of passing it into `run(…)`, this configuration can also be passed in the constructor of `SpringApplication`—this leads to three variants:

- `SpringApplication(Class<?>... primarySources)`
- `static ConfigurableApplicationContext run(Class<?> primarySource, String... args)`
- `static ConfigurableApplicationContext run(Class<?>[] primarySources, String[] args)`

The call of the static `run(…)` method with only a single start configuration has been selected by the Initializr:

```
@SpringBootApplication
public class Date4uApplication {
  public static void main( String[] args ) {
    SpringApplication.run( Date4uApplication.class, args );
  }
}
```

Listing 2.5 Date4uApplication.java

In the `run(…)` method, the Initializr has used the `Class` object of `Date4uApplication`. That means, our own class is used as the start configuration. But the start configuration was the class annotated with `@SpringBootApplication`, and `@SpringBootApplication` is nothing more than a summary of, to put it simply, three annotations:

```
@SpringBootConfiguration
@EnableAutoConfiguration
@ComponentScan
```

The listed annotation `@SpringBootConfiguration` is in turn a specialization of `@Configuration`, and that in turn is an `@Component`. Components are automatically recognized. And the fact that `@SpringBootApplication` is annotated with `@ComponentScan` is important, and we want to take a closer look at that.

@ComponentScan + @Configuration

Annotation `@ComponentScan` can be used in combination with annotation `@Configuration`. Then, the package containing the class annotated with `@ComponentScan` and `@Configuration`, including all subpackages, is searched for corresponding components via classpath scanning.

The class with the `main(…)` method and the start configuration can be separated without any problems. The following would also be fine (`Date4uApplication` is supposed to be in the default package as an example):

```
// not annotated with @SpringBootApplication
public class Date4uApplication {
  public static void main( String[] args ) {
    SpringApplication.run( Date4uConfiguration.class, args );
  }
}
```

The startup configuration can be found in the `com.tutego.date4u` package:

```
package com.tutego.date4u;

@SpringBootApplication
class Date4uConfiguration { }
```

The `run(…)` method gets the startup configuration `Date4uConfiguration`, which is annotated with `@SpringBootApplication`. This contains `@ComponentScan` and `@Configuration`, so the classpath scanning starts as previously at `com.tutego.date4u`.

@ComponentScan Annotation Type

The declaration of `@ComponentScan` looks like this:

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
@Documented
```

```
@Repeatable(ComponentScans.class)
public @interface ComponentScan {

  @AliasFor("basePackages") String[] value() default { };
  @AliasFor("value") String[] basePackages() default { };

  Class<?>[] basePackageClasses() default { };

  boolean useDefaultFilters() default true;

  Filter[] includeFilters() default { };
  Filter[] excludeFilters() default { };

  String resourcePattern() default …

  Class<? extends BeanNameGenerator> nameGenerator() default …
  Class<? extends ScopeMetadataResolver> scopeResolver() default …
  ScopedProxyMode scopedProxy() default …
  boolean lazyInit() default false;

  @Retention(RetentionPolicy.RUNTIME) @Target({})
  @interface Filter { … }
}
```

We'll see in a moment how to define the start packages and how to include or exclude types using filters. The `@ComponentScan` type still has a nested annotation type called `@Filter`.

basePackages

If we use `@ComponentScan` and don't use annotation attributes at all, by default, classpath scanning will start in the package where this annotated class is located. The package including all subpackages will then be included.

However, we can specify the starting package or several starting packages where classpath scanning should start. This can be solved with annotation attribute `basePackages`. With `basePackages`, an array of base packages can be set; different notations are allowed. The following configuration results in the same recognized types if `com.tutego.date4u.core` and `com.tutego.date4u.interfaces` are the roots:

```
@Configuration
@ComponentScan( basePackages = { "com.tutego.date4u.core",
                                 "com.tutego.date4u.interfaces"} )
```

The attributes `basePackages` and `value` are aliases. That is, `basePackages` can be omitted for an annotation attribute:

```
@Configuration
@ComponentScan( { "com.tutego.date4u.core",
                  "com.tutego.date4u.interfaces"} )
```

The packages can also be in a string, with whitespace, commas, or semicolons allowed as separators:

```
@Configuration
@ComponentScan( "com.tutego.date4u.core, com.tutego.date4u.interfaces" )
```

A combination is also possible, that is, an array of strings with multiple package names:

```
@Configuration
@ComponentScan( { "com.tutego.date4u.core com.tutego.date4u.interfaces" } )
```

With `basePackages`, any packages can be included and added, even those that are in a completely different package hierarchy. However, then the package in which the annotated `@ComponentScan` type is located is no longer taken as the root package, but only the packages named at `basePackages`.

basePackageClasses

The problem with `basePackages` are the strings. These quickly cause problems when refactoring. That's why there is an alternative way—`basePackageClasses`. Then, a type token is specified. Here's an example:

```
@Configuration
@ComponentScan( basePackageClasses = { A.class, B.class } )
```

Again, `@Configuration` and `@ComponentScan` occur together, and the array specifies that classpath scanning should start at the exact packages where types `A` and `B` are located.

Of course, `A` and `B` could be any types from these packages, but this is also risky: What if classes `A` and `B` are moved to another package? Then the classpath scanning might scan the wrong packages.

The problem can be solved with empty types in the desired packages, which `basePackageClasses` then references. As an example, let's assume that the classpath scanning should detect all components under `com.tutego.date4u`. Then, an empty interface will be defined in this package:

```
package com.tutego.date4u.core;
public interface CoreModule { }
```

Here, the interface is called `CoreModule`. This exact type is then referenced in `basePackageClasses`:

```
@ComponentScan( basePackageClasses = { CoreModule.class } )
```

If refactoring is done later and special types move to other packages, then this "module interface" isn't moved and remains in the correct place.

includeFilters

With `basePackages` and with `basePackageClasses`, it's possible to include all types that are in certain packages. Filters control more precisely which types should be included in the packages. For the filters, there is the possibility to either include types or exclude types; for this, `@ComponentScan` declares two annotation attributes: `includeFilters` to selectively include types and `excludeFilters` to remove types.

Because by default `@ComponentScan` includes all `@Component` types, you can disable automatic detection for `includeFilters` with an annotation attribute: `useDefaultFilters = false`. In that case, only what is included by the `includeFilters` will be used.

@ComponentScan.Filter

`includeFilters` and `excludeFilters` are of type `@ComponentScan.Filter[]`. The annotation type `@Filter` is nested in annotation type `@ComponentScan`. A filter contains additional information, for example, what kind of filter can be used. Filters can be constructed according to a whole range of criteria, for example:

- What is type-compatible?
- What fits on a regular expression (regex)?
- What annotations do the types have (default setting)?

In this example of the first filter type, the classpath scanner should be configured to detect only components of type `Thumbnail`:

```
@Configuration
@ComponentScan(
    // basePackages = "com.tutego.date4u",
    useDefaultFilters = false,
    includeFilters = @ComponentScan.Filter(
        type = FilterType.ASSIGNABLE_TYPE,
        classes = Thumbnail.class )
)
class ThumbnailScanner { }
```

The annotation attribute `basePackages` could be set, but it doesn't have to be because without specification, the starting package of the search is the package where `ThumbnailScanner` is located. If it should not be in `com.tutego.date4u`, and we want other starting packages, we set `basePackages`.

The assignment `useDefaultFilters = false` controls in my example that not every `@Component` component is automatically detected and logged in. `FilterType.ASSIGNABLE_TYPE` sets the actual criteria that all `thumbnail` implementations are found.

It's useful to leave `useDefaultFilters = true` (this is the default) and still work with an `includeFilters`. The reason is that `includeFilters` also finds types that aren't annotated with `@Component` and still registers them as a Spring-managed bean.

excludeFilters

Let's formulate a second example with `excludeFilters`. We *don't* want to include types that extend `AwtBicubicThumbnail`. `useDefaultFilters` remains `true`, so we can assume an entire list of all components.

```
@Configuration
@ComponentScan(
    // basePackages = "com.tutego.date4u",
    excludeFilters = @ComponentScan.Filter(
        type = FilterType.ASSIGNABLE_TYPE,
        classes = AwtBicubicThumbnail.class )
)
class ThumbnailScanner { }
```

Currently there is only one type in `excludeFilters`, but multiple types are possible in the array notation. The found types pass through this "filter," and if they are `instanceof AwtBicubicThumbnail`, they *aren't* included in the list.

Example for includeFilters with FilterType.ANNOTATION

The next example combines an `includeFilters` together with `FilterType.ANNOTATION`. Only types annotated with `ThumbnailRenderingFast` are recognized. All other classes should be ignored; therefore, `useDefaultFilters = false`:

```
@Configuration
@ComponentScan(
    useDefaultFilters = false,
    includeFilters = @ComponentScan.Filter(
        // type = FilterType.ANNOTATION,
        classes = { ThumbnailRenderingFast.class } )
)
class FastThumbnailScanner { }
```

If `useDefaultFilters` were `true` (the default), we would have many more types because `includeFilters` *expands* the search. Because filter type `ANNOTATION` is the default, the assignment isn't necessary. With `classes`, the exact annotations are listed, which the found classes should have.

The code can be written a little more compactly because `classes` is an alias for `value`, and you can also omit the curly braces for an array with only one element. From `ComponentScan.Filter`, the filter can be imported statically with possible abbreviations as follows:

```
@Configuration
@ComponentScan(
    useDefaultFilters = false,
    includeFilters    = @Filter( ThumbnailRenderingFast.class ) )
class FastThumbnailScanner { }
```

Example of [include|exclude]Filters with FilterType.REGEX

Let's look at another example, using `includeFilters` and `FilterType.REGEX` for regex, along with `excludeFilters`. The following example is intended to find all types ending in "New" or "Old", but ignore `LegacyAwtThumbnailOld`:

```
@Configuration
@ComponentScan(
    useDefaultFilters = false,
    includeFilters = @Filter(
        type = FilterType.REGEX, pattern = ".*(New|Old)" ),
    excludeFilters = @Filter(
        type = FilterType.ASSIGNABLE_TYPE,
        classes = LegacyAwtThumbnailOld.class )
)
class FastThumbnailScanner { }
```

With `useDefaultFilters = false`, there are no `@Component` components by default. `includeFilters` selects via `type = FilterType.REGEX` all type names that match `".*(New|Old)"`. This is the typical notation of Java regex. There are also Ant-style path pattern matchers in the Java universe; these aren't used here, but instead classic regex according to the rules described by `java.util.regex.Pattern`. Next, we add an `excludeFilters` with the `ASSIGNABLE_TYPE`. Of course, we could have worked with a `FilterType.REGEX` in the example with this same type name, but strings aren't very robust when refactoring. If we can express something as a `Class` object, all the better.

FilterType.CUSTOM

If the mentioned filters can't cover a scenario, a completely independent filter can be implemented. The first thing to do is to implement the `TypeFilter` interface, as briefly outlined here:

```
class MyTypeFilter implements TypeFilter {
  @Override
  public boolean match( MetadataReader metadataReader,
                        MetadataReaderFactory metadataReaderFactory ) {
    ClassMetadata classMetadata = metadataReader.getClassMetadata();
    …
  }
}
```

The `TypeFilter` interface is a functional interface with a `match(…)` method. It must decide whether a type will later appear in the result list or not. For the method to make the decisions, the framework passes two pieces of information: `MetadataReader` and `MetadataReaderFactory`. With the types, the method can find out more about the type that was just scanned, or it can inquire something from all other components.

The custom type filter implementation is referenced later at `@ComponentScan`:

```
@Configuration
@ComponentScan(
    useDefaultFilters = false,
    includeFilters = @Filter(
        type = FilterType.CUSTOM,
        classes = MyTypeFilter.class )
)
class MyTypeFilterScanner { }
```

In this way, a filter can test arbitrary things, for example, whether a certain class contains certain methods or certain variables.

In this section, we've looked in detail at configuring classpath scanning. The rules can get confusing, and the recommendation is to keep it simple. It's best to use only the startup packages for classpath scanning because this is the easiest to maintain.

2.4 Interactive Applications with the Spring Shell

Spring Shell (*https://github.com/spring-projects/spring-shell*) is an open-source side project that makes it easy to write interactive shell applications.

2.4.1 Include a Spring Shell Dependency

For the Spring Shell project, there is a Spring Boot Starter, and, in principle, the Shell project can also be selected in the Initializr. Because we've already created our Date4u project, we declare a property for a version number and add the dependency manually in our POM file:

```
<properties>
  <!-- … -->
  <spring-shell.version>3.1.3</spring-shell.version>
</properties>
```

Listing 2.6 pom.xml Extension 1

```
<dependency>
  <groupId>org.springframework.shell</groupId>
  <artifactId>spring-shell-starter</artifactId>
  <version>${spring-shell.version}</version>
</dependency>
```

Listing 2.7 pom.xml Extension 2

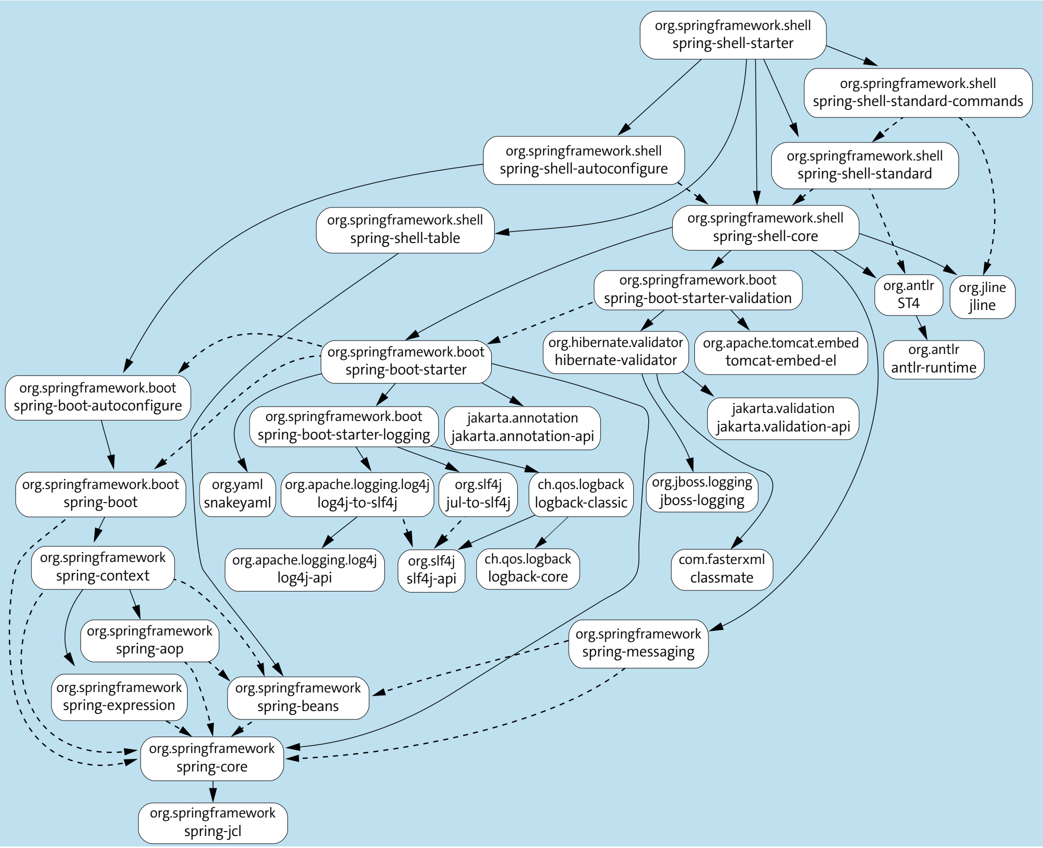

org.springframework.shell spring-shell-starter
org.springframework.shell spring-shell-standard-commands
org.springframework.shell spring-shell-autoconfigure
org.springframework.shell spring-shell-standard
org.springframework.shell spring-shell-table
org.springframework.shell spring-shell-core
org.springframework.boot spring-boot-starter-validation
org.antlr ST4
org.jline jline
org.springframework.boot spring-boot-starter
org.hibernate.validator hibernate-validator
org.apache.tomcat.embed tomcat-embed-el
org.antlr antlr-runtime
org.springframework.boot spring-boot-autoconfigure
org.springframework.boot spring-boot-starter-logging
jakarta.annotation jakarta.annotation-api
jakarta.validation jakarta.validation-api
org.springframework.boot spring-boot
org.yaml snakeyaml
org.apache.logging.log4j log4j-to-slf4j
org.slf4j jul-to-slf4j
ch.qos.logback logback-classic
org.jboss.logging jboss-logging
org.springframework spring-context
org.apache.logging.log4j log4j-api
org.slf4j slf4j-api
ch.qos.logback logback-core
com.fasterxml classmate
org.springframework spring-aop
org.springframework spring-messaging
org.springframework spring-expression
org.springframework spring-beans
org.springframework spring-core
org.springframework spring-jcl

Figure 2.3 Dependencies from the Spring Shell Starter Project

2.4.2 First Interactive Application

If you start the Spring Boot application, the mere existence of this dependency causes the shell to be started:

```
shell:>
```

The Spring Shell project employs a runner to implement the shell, which, similar to other runners, is triggered automatically upon container startup. This mechanism allows the program to transition into the interactive application. However, `SpringApplication.run(…)` “blocks” the application: it only proceeds to execute the code statements following `run(…)` once the shell is closed.

The shell comes with integrated commands, which can be viewed by using the `help` command:

```
shell:>help
AVAILABLE COMMANDS

Built-In Commands
       help: Display help about available commands
       exit: Exit the shell.
       stacktrace: Display the full stacktrace of the last error.
       clear: Clear the shell screen.
       quit: Exit the shell.
       history: Display or save the history of previously run commands
       version: Show version info
       script: Read and execute commands from a file.
shell:>exit
```

If we enter `help`, the shell shows built-in standard commands, which can also be disabled with a configuration property. With `exit`, we can terminate the shell, which then also terminates our program.

2.4.3 Write Shell Component: @ShellComponent, @ShellMethod

New commands can be registered with the shell. The commands are set in classes annotated with `@ShellComponent`. Our task is to create a new shell command by implementing a shell component, and we’ll refer to this component as `FsCommands` class.

```
import org.springframework.shell.standard.*;

@ShellComponent
public class FsCommands {

  @ShellMethod( "Display required free disk space" )
  public long minimumFreeDiskSpace() {
    return 1_000_000;
  }
}
```

Listing 2.8 FsCommands.java

The purpose of the `FsCommands` class is to execute commands on the file system. To do this, the class is annotated with `@ShellComponent`. `@ShellComponent` is itself an `@Component`:

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
@Documented
@Component
public @interface ShellComponent {

  /**
   * Used to indicate a suggestion for a logical name for the component.
   * @return the suggested component name, if any
   */
  String value() default "";
}
```

This means that when we later launch our Spring Boot applications, this shell component will also be recognized as a Spring-managed bean.

Simple Shell Methods

In an `@ShellComponent` class, you put shell methods that users can call from the outside. There are three requirements for this:

- **Naming**
  A shell method has a very special name because this is automatically converted into the shell command. Method `minimumFreeDiskSpace()` will return the number of bytes that our application (fictitiously) needs minimally. Later, the command is available through `minimum-free-disk-space`. Of course, the method name can also be decoupled from the command name.
- **`@ShellMethod`**
  A shell method is annotated.
- **Documentation**
  A shell method gets a documentation, which is displayed later with the help.

Shell methods can return arbitrary values, and these values are then printed by the shell in the string representation.

If the class is written and registered as a Spring-managed bean, it will appear in the help after the program is restarted, and the commands can be called:

```
shell:>help
AVAILABLE COMMANDS

Built-In Commands
       help: Display help about available commands
       exit: Exit the shell.
       stacktrace: Display the full stacktrace of the last error.
       clear: Clear the shell screen.
       quit: Exit the shell.
       history: Display or save the history of previously run commands
       version: Show version info
       script: Read and execute commands from a file.

Fs Commands
       minimum-free-disk-space: Display required free disk space


shell:>minimum-free-disk-space
1000000
shell:>exit
```

For the call on the shell, the camel case notation must be translated from the method name to kebab case, so `minimumFreeDiskSpace` becomes the command name `minimum-free-disk-space`. After the user input, the expected output 1000000 appears.

Shell Methods with Parameters

Shell methods can accept values from the outside. Suppose a shell method is to convert a string to lowercase:

```
@ShellMethod( "Convert to lowercase string" )
public String toLowercase( String input ) { return input.toLowerCase(); }
```

If you call such a shell method without parameters, there is an error:

```
shell:>to-lowercase
Missing mandatory option --input.
```

With one parameter, it gives the desired answer:

```
shell:>to-lowercase ABC
abc
```

If we want to set a string with whitespace, the string must be enclosed in single or double quotes; otherwise, the shell interprets every string separated by whitespace as an additional parameter, but method `toLowercase(String)` expects only exactly one string.

```
shell:>to-lowercase Mr. Bean
mr.
shell:>to-lowercase "Mr. Bean"
mr. bean
shell:>to-lowercase 'Mr. Bean'
mr. bean
```

In the shell, the arguments aren't separated with commas as we do in the Java program, but the parameters are simply separated with the space character.

Shell Component Accesses FileSystem

The shell class should get a second method that returns the free disk space in gigabytes. The additions are highlighted.

```
@ShellComponent
public class FsCommands {

  private final FileSystem fs = new FileSystem();

  // …

  @ShellMethod( "Display free disk space" )
  public String freeDiskSpace() {   // free-disk-space
    return DataSize.ofBytes( fs.getFreeDiskSpace() )
                   .toGigabytes() + " GB";
  }
}
```

Listing 2.9 FsCommands.java Extension

Method `freeDiskSpace()` returns a string with the number of free bytes, converted to gigabytes (GB). The free bytes are retrieved via the `FileSystem` object. The Spring class `DataSize` helps to convert the bytes into gigabytes.

Configurations

Configuration properties can control further behavior of the Spring Shell:

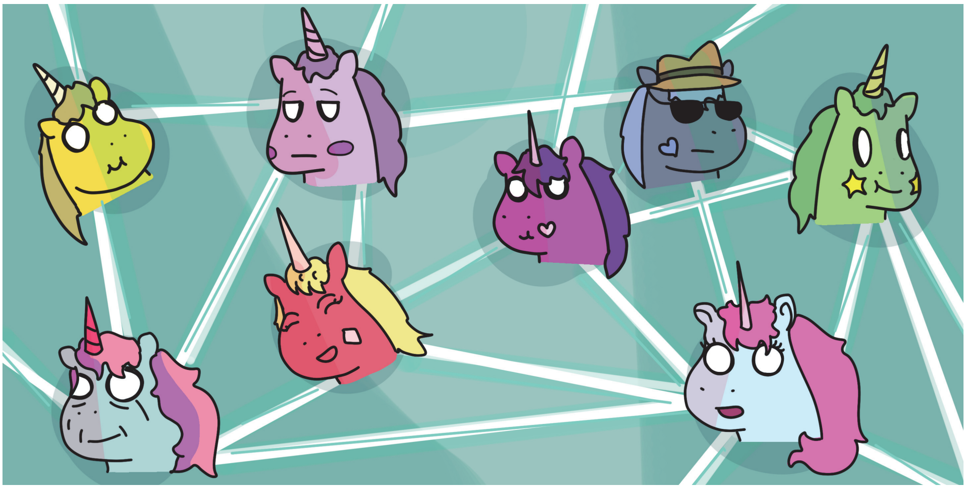
- If `spring.shell.interactive.enabled` is set to `false`, the interactive shell won't be started, even if it's present as a dependency in the POM file. This is absolutely essential for test cases because JUnit test cases are never interactive.
- By default, Spring Shell writes all entered commands to a log file. If you set `spring.shell.history.enabled` to `false`, the file isn't written.
- The name of the log file can be set via `spring.shell.history.name`; if it's not set, `spring.application.name` is evaluated. If this is also not set, the name is automatically *spring-shell.log*.

2.5 Injecting Dependencies

In this section, we'll look at a specialty of the Spring Framework: injection. Injection is about getting the required references to the appropriate places. This is a central mechanism of the Spring Framework.

2.5.1 Build Object Graphs

Every application consists of a mesh of objects that know each other. This is because an object can never do everything on its own, nor should it be able to. This is one of the core principles of object orientation, the single-responsibility principle: an object can do one thing really well and also only performs exactly one task. If there is a request that the object itself can't realize, it turns to another object, which in turn does the task perfectly well.



For example, `FileSystem` could be used by two different components: to implement shell commands in the `FsCommands` class and in a `PhotoService` that wants to load or save images using the file system (see Figure 2.4).

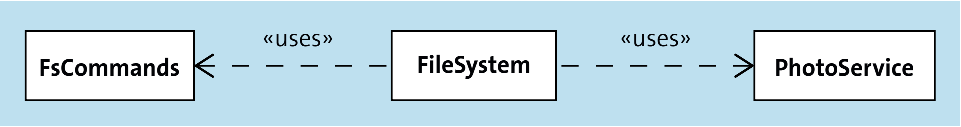


Figure 2.4 "FileSystem" Used by "FsCommands" and "PhotoService"

The central question is how these objects come together because this object graph must be built one day. In traditional software, if an object is needed, there are two ways to get these references: one way is to build the object yourself with `new`, and the other way is to let factories create the instances. The advantage of factory methods is that they contain code and can dynamically determine the type at runtime. Factories, factory methods, or abstract factories are design patterns, and configuration usually takes place in code. But Spring Framework can do the job of object creation for us.

Here's a little flashback: `FsCommands` did use `FileSystem` for the implementation of the one command:

```
@ShellComponent
public class FsCommands {

  // // // // // // // // // // // // // // // //
  private final FileSystem fs = new FileSystem();
  // // // // // // // // // // // // // // // //

  @ShellMethod( "Display free disk space" )
  public String freeDiskSpace() {   // free-disk-space
    return DataSize.ofBytes( fs.getFreeDiskSpace() ).toGigabytes() ) + " GB";
  }

  …
}
```

The program itself builds its own instance of the `FileSystem` at this point with `new`. But is that wise? There are disadvantages:

- For example, if a second component (e.g., the `PhotoService`) also has `new FileSystem()`, then two instances of the `FileSystem` are created. This is unnecessary because the file system should be a singleton.
- With `new`, the object type is explicit, and it's not so easy to change it. If the code is `new FileSystem()`, we always get an instance of a `FileSystem` and nothing else. And the constructor that contains the initialization routine will always initialize a `FileSystem`. As a result, the concrete object and the initialization routine are hardwired into the code. If a special implementation of a `FileSystem`, that is, a subclass, is desired, we must use factories. Even if we want to put the `FileSystem` as mock objects in a `FsCommands` for testing, we have to prepare that by hand.

From these disadvantages, the following desires arise:

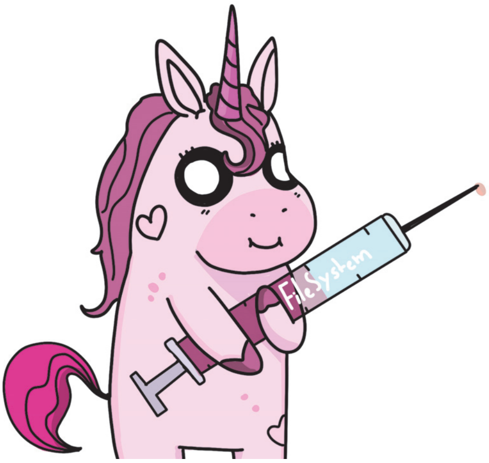
- The objects are used as singletons; that is, several `FileSystem` copies should not to be created.
- There should be a simple way to change the concrete type.

The problem could be solved by getting away from instantiating ourselves with `new` and letting someone else do the instantiating.

2.5.2 Inversion of Control and Dependency Injection

A central concept of Spring is the *inversion of control* (IoC). But what do we mean by "control," and what is being "inverted" here? Normally, it's our own programs that have control over what they need. Programs open files and read configurations, create objects they need, and so on. IoC means that this control is reversed, and the action comes from somewhere else. An object is no longer created by us, but it's built elsewhere. Or, the content of configuration files is necessary: it's no longer our own program that reads the configurations, but the program relinquishes control so that another place reads the file. That is, if we normally build up the object graph ourselves, we now leave it to a magic instance from outside to build up these objects and to give them to us later.

In this context, a second term becomes interesting, which is *dependency injection*. This is a special form of IoC. Dependency injection means that the dependencies are *injected* so that the component can actually accept and get those references. This is because when a program inverts control and no longer builds objects itself, the references must come back to the program. In other words, with dependency injection, these dependencies are injected into components. The term goes back to architect Martin Fowler, who coined it around 2004.[8] This is also the time when the Spring container was created.



2.5.3 Injection Types

With dependency injection, someone has to do this injection. In principle, this can be done by hand—and in fact will be done by hand in some cases later in a JUnit test case—but the Spring Framework can do this for us.

Let's take the `FsCommands` class: an object of type `FileSystem` was needed, and `FsCommands` built the instance itself with `new`:

---

[8] *http://martinfowler.com/articles/injection.html*

```
public class FsCommands {
  private final FileSystem fs = new FileSystem();
  …
}
```

If the Spring Framework can create and inject the `FileSystem`, how does the reference get “inside”? Ultimately, there are three possibilities for dependency injection:

- **Constructor injection**
  The container calls the constructor on its `@Component` classes, and a Spring-managed bean is created. A constructor can “wish” a reference to another object that the component has a dependency on. If we’ve implemented a parameterless constructor in `FsCommands` in the past, a parameterized constructor can request that reference now, and the container must then provide that reference because otherwise the constructor can’t be called. Our own constructor can accept and store this reference.
- **Setter injection**
  The container calls a setter that stores the reference for later.
- **Field injection**
  The Spring container modifies an instance variable directly.

What the container does is called *wiring*. When the container starts up, it will initialize all the Spring-managed beans, resolve the required dependencies, and do the wiring. Later, when the `run(…)` method starts the container, there will be a completely initialized object graph in memory.

Next, we’ll look at these three dependency injection options in more detail.

FileSystem in FsCommands with Constructor Injection

Constructor injection is typically considered the most crucial type of injection in practice, so let’s begin with it:

```
@ShellComponent
public class FsCommands {

  // private final FileSystem fs = new FileSystem();
  // ->

  private final FileSystem fs;

  @Autowired
  public FsCommands( FileSystem fs ) { this.fs = fs; }

  …
}
```

We leave the line with `new` as a comment.

First, a parameterized constructor is introduced that “wants” a reference to the `FileSystem` from the Spring Framework when it’s instantiated. `FsCommands` can’t be built if a `FileSystem` hasn’t been built first. This implies a certain order in which the container must build the objects. It must order them according to usage. If `FsCommands` is to be built up, a `FileSystem` must have been built up before because otherwise the parameterized constructor can’t be called by the framework at all.

An injection takes place by default only if the following are true:

- `FileSystem` is a Spring-managed bean.
- The target being injected into is also a component.

If `FsCommands` wasn’t a component, Spring would not consider this class. Because `@ShellComponent` is a special `@Component`, it’s part of the object graph.

The container must know where to inject something. It doesn’t do this on its own, and certain hints are needed as to where wiring is desired. In the preceding example, annotation `@Autowired` gives this hint. `@Autowired` can be omitted in some cases.

FileSystem in FsCommands with Setter Injection

The next injection variant is the *setter injection*:

```
@ShellComponent
public class FsCommands {

  // private final FileSystem fs = new FileSystem();
  // ->

  private FileSystem fs;

  @Autowired
  public void setFileSystem( FileSystem fs ) { this.fs = fs; }
  …
}
```

This time, the setter is annotated with `@Autowired` so the container knows which method to call. It would be foolhardy for the container to call every method where a `FileSystem` is required in the parameter. That’s why `@Autowired` specifically marks the setter that the container should call. The body of the setter looks exactly like the body of the constructor: A `FileSystem` is received and stored in the variable `fs`. Any number of setters can be `@Autowired`.

One difference between constructor injection and setter injection can already be seen: with constructor injection, the described instance variable can be `final`, which doesn’t work with a setter.

Note

With setter injection, the method doesn't have to start with the prefix `set`. Spring can inject Spring-managed beans into any method annotated with `@Autowired`.

FileSystem in FsCommands with Field Injection

Now, let's proceed to the final type of injection—*field injection*:

```
@ShellComponent
public class FsCommands {

  // private final FileSystem fs = new FileSystem();
  // ->

  @Autowired private FileSystem fs;

  …
}
```

If the program needs a reference in an instance variable, the variable is annotated with `@Autowired`. Field injection is nice and short, and it's even less source code than with the `new` variant. The reference is automatically stored into this instance variable via reflection. For static variables, `@Autowired` doesn't work.

The variable can't be `final`, as with setter injection because auto-wiring occurs a bit later, after the constructor has already been called. Any number of instance variables can be `@Autowired`.

The options can be mixed. Thus, a parameterized constructor, parameterized setter, and annotated instance variables are possible without any problems—obviously for different Spring-managed beans. However, in principle, the same Spring-managed bean can also be introduced in the three ways.

What Is the best Injection?

Whenever there are multiple ways to do something, the question of the best option inevitably arises. Following are some conclusions for your orientation:

- Constructor and setter injection are good for testing because we can build the object graph ourselves. With field injection, we usually have private instance variables. This means that for tests, we need to work with reflection, which means that we need to know the inner workings of the program. That's where using constructor or setter injection is much cleaner. Constructor injection has the advantage of making it explicitly clear which references are needed by the Spring-managed bean. Setter injection and even more so field injection hides these dependencies. In the API documentation, the constructor injection makes it very easy to see what references are needed to fully initialize the object. And if you see a very large parameter list, then that is a strong indication of an antipattern, called a *God object* (can do everything, knows everyone). Private instance variables described via field injection aren't visible from the outside, which can be irritating for tests of the class. That means, according to the API documentation ,the class has a simple constructor, but later in use, it can continuously crash with `NullPointerException` because references weren't set.

- Constructor injection fits well with immutable data types, that is, data types with states that are initialized once but don't change later. With constructor injection, if a constructor stores the reference, the instance variables can easily be final. This doesn't work with setter injection and field injection: you can't make these variables final because this initialization takes place at a much later stage. For this reason, constructor injection is an excellent choice for objects that aren't allowed or not required to make any changes to their object states later on.
- If setters or constructors receive objects during wiring, they can directly become a source for other objects, which are then stored internally for later use. For example, a `Path` is requested for a file location, but instead of storing this path internally, a file is loaded directly via the path, and the file contents are stored internally.
- Circular dependencies don't work with constructor injections. But this isn't a big problem because such cases should be avoided. If interested, you can find more details in Section 2.5.8.
- Constructors and setters are usually `public`, but instance variables are usually `private`. If we want to test this component, we need to describe the variables via reflection.

Note

In some places in the book, field injection is used to keep the examples short. For test cases, however, field injection is the norm.

Close FsCommands for Now

We can close our `FsCommands` class with constructor injection for the time being.

```
package com.tutego.date4u.interfaces.shell;

import …

@ShellComponent
public class FsCommands {

  private final FileSystem fs;
  public FsCommands( FileSystem fs ) {
    this.fs = fs;
  }

  @ShellMethod( "Display free disk space" )
  public String freeDiskSpace() {   // free-disk-space
    return DataSize.ofBytes( fs.getFreeDiskSpace() ).toGigabytes() + " GB";
  }
}
```

Listing 2.10 FsCommands.java

When the Spring container boots up, it will use reflection to build an object of type `FileSystem` and also call the parameterized constructor of `FsCommands` and pass the reference to the `FileSystem` bean. This concludes our `FsCommands` class for now.

Tip

Constructors don't necessarily have to be annotated with `@Autowired` for wiring. However, `@Autowired` can contribute to better documentation, although this is a bit of a matter of taste. If there is more than one parameterized constructor and thus ambiguities, the intended constructor must be annotated with `@Autowired`.

Tip: IntelliJ Tip

If a program uses field injection, the integrated development environment (IDE) undercurls the location and offers to rewrite the code into a constructor injection.

2.5.4 PhotoService and PhotoCommands

In our application, we want to add two new components: `PhotoService` that can load and save photos, and `PhotoCommands` so that we can also upload and download photos and view metadata of photos on the shell (see Figure 2.5).

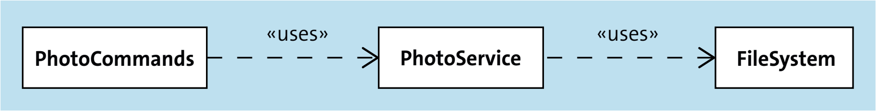


Figure 2.5 A Command Class Using a Service That's Using Another Service

The `PhotoService` looks like this:

```
@Service
public class PhotoService {
  private final FileSystem fs;

  public PhotoService( FileSystem fs ) {
    this.fs = fs;
  }

  public Optional<byte[]> download( String name ) {
    try { return Optional.of( fs.load( name + ".jpg" ) ); }
    catch ( UncheckedIOException e ) { return Optional.empty(); }
  }
}
```

An image name is passed to the `download(String)` method, and `PhotoService` appends the suffix `.jpg` to the file name and delegates loading to the `FileSystem`. Because the `FileSystem` throws an exception if the image doesn't exist, the program catches this exception and returns `Optional.empty()`; otherwise, the byte array is wrapped in `Optional` and returned. An IO stream (or `org.springframework.core.io.Resource`) would be the right choice in a "real" implementation.

The second component is `PhotoCommands`. This class is supposed to implement commands that have something to do with photos, for example, displaying the metadata of a photo and later uploading photos.

```
@ShellComponent
public class PhotoCommands {

  private final PhotoService photoService;

  public PhotoCommands( PhotoService photoService ) {
    this.photoService = photoService;
  }

  @ShellMethod( "Show photo" )
  String showPhoto( String name ) {// show-photo
    return photoService.download( name ).map( bytes -> {
      try {
        var image = ImageIO.read( new ByteArrayInputStream( bytes ) );
        return    "Width: "     + image.getWidth()
              + ", Height: " + image.getHeight();
      } catch ( IOException e ) { return "Unable to read image dimensions"; }
    } ).orElse( "Image not found" );
  }
}
```

Listing 2.11 PhotoCommands.java

The method `showPhoto(String)` is given the name of the image and thus delegates to the `PhotoService`. We get `optional` back from the `download(…)` method, which may or may not contain the photo. If there is a photo, the `map(…)` method will map that photo to a string with the metadata. The Java SE class `ImageIO` helps to load the image (`BufferedImage`), and we get the metadata directly from this `BufferedImage`. Because `ImageIO.read(…)` can throw a checked exception, this exception is caught and translated into the error message **Unable to read image dimensions**. If the file could not be loaded, error message **Image not found** is passed to the shell.

We've built a nice chain: `PhotoCommands` wants the `PhotoService`, and the `PhotoService` wants the `FileSystem`. When wiring, the Spring Framework will build a `FileSystem` first because it's the first object needed so that the reference in the constructor can be passed by the `PhotoService`; the `PhotoService` then comes into the context as a Spring-managed bean. With `PhotoCommands`, the same thing happens: the constructor wants a `PhotoService`. In the end, Spring has validly constructed the object graph. Of the three options, field injection, setter injection, and constructor injection, the program uses constructor injection because that is the best variant of injection in everyday use.

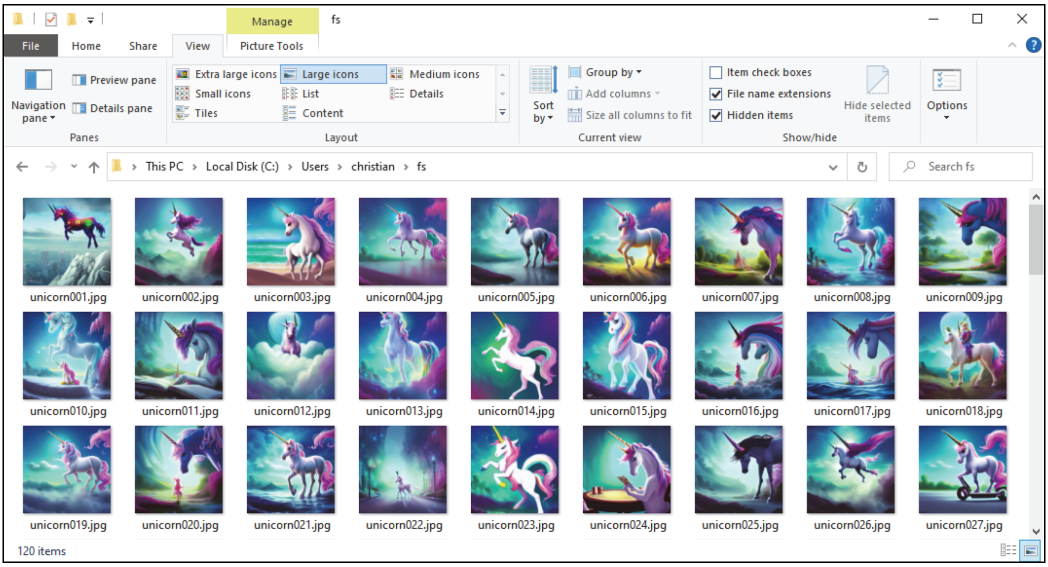

Manage
fs
File
Home
Share
View
Picture Tools
Navigation pane
Preview pane
Details pane
Panes
Extra large icons
Large icons
Medium icons
Small icons
List
Details
Tiles
Content
Layout
Sort by
Group by
Add columns
Size all columns to fit
Current view
☐ Item check boxes
☑ File name extensions
☑ Hidden items
Hide selected items
Show/hide
Options
This PC › Local Disk (C:) › Users › christian › fs
Search fs
unicorn001.jpg
unicorn002.jpg
unicorn003.jpg
unicorn004.jpg
unicorn005.jpg
unicorn006.jpg
unicorn007.jpg
unicorn008.jpg
unicorn009.jpg
unicorn010.jpg
unicorn011.jpg
unicorn012.jpg
unicorn013.jpg
unicorn014.jpg
unicorn015.jpg
unicorn016.jpg
unicorn017.jpg
unicorn018.jpg
unicorn019.jpg
unicorn020.jpg
unicorn021.jpg
unicorn022.jpg
unicorn023.jpg
unicorn024.jpg
unicorn025.jpg
unicorn026.jpg
unicorn027.jpg
120 items

Figure 2.6 Images of Unicorns in the Local fs Directory

Note

The visibility of the injection point doesn't matter. That is, it doesn't matter whether we inject into a public constructor, a private constructor, a public instance variable, or a private one (the same is true for setters). We should stay away from private parameterized constructors because private generally means the following: only the class itself may use the constructor (perhaps through a static factory method), but the initialization isn't part of the public interface. This is callable by reflection anyway because private members can also be addressed via reflection. Tests would also have to use mandatory reflection for private injection points. Package-visible constructors are preferable because the test classes are usually in the same package, and the constructors can then be called directly.

If you want to use the photo commands, you should prepare an image in the local file system. Let's navigate to the user directory—there our Java software has previously created a directory *fs*, which is the root directory for our images—more precisely, for everything that the `FileSystem` can load and store.

If you copied the images, you'll find a photo with the file name *unicorn001.jpg* under *fs*. Then the image name is `unicorn001` (without file extension). With the programmed command `show-photo`, we can ask for the metadata. After starting the program, the output should look like this:

```
shell:> show-photo unicorn001
Width: 512, Height: 512
```

To sum up, Spring automatically created and injected various objects for us. The `PhotoCommands` component accesses the `PhotoService`. `PhotoService` needs the `FileSystem`. We also had another user of `FileSystem`, which was `FsCommands`. By default, the components are built only once and are singletons.

Note

As part of the book download—and alternatively at *https://github.com/ullenboom/120-unicorn-photos*—there are 120 pictures of unicorns, named *unicorn001.jpg* to *unicorn120.jpg*. It's recommended for the tasks and exercises to put the images into your own *fs* directory.

2.5.5 Multiple Dependencies

Often, a component has multiple collaborators. That is, a component needs references to various other components to fully work. Of course, multiple references can be injected into a Spring-managed bean. There are several ways to do this:

- If you proceed via constructor injection, you simply have a constructor with multiple parameters.
- When working with field injection, there are several instance variables annotated with `@Autowired`.
- A JavaBean setter is a method with the prefix `set` and exactly one parameter. This is how we've encountered setter injection so far. However, it doesn't have to be that way because the method can have any name and request any number of Spring-managed beans in the parameter list. A parameterized method is basically nothing

more than a parameterized constructor—only the constructor carries the name of the class, and an `@Autowired` method can carry any name.

Here's a fictional example:

```
@Autowired
void setup( FileSystem fs, PhotoService photoService ) {
 …
}
```

The `setup(…)` method requests a `FileSystem` and a `PhotoService`. Such constructions can be handy because while a classic setter has only a single parameter, with this setup, you can wish for several things afterwards, including optional Spring-managed beans.

2.5.6 Behavior in Case of a Missing Component

A wiring always means, no matter if we realize it via constructor injection, setter injection, or field injection, that we wish a component. This component must not be missing there, otherwise there will be an error message, and the container won't start up. This is an important principle in Spring: if you survive the `run(…)` method to start the container, all components were found and could be initialized.

Suppose we have `@Autowired` on a `String` that should not exist like this:

```
@Autowired String string;
```

Launching the application isn't possible and leads to an error:

```
***************************
APPLICATION FAILED TO START
***************************

Description:

Field string in […] required a bean of type 'java.lang.String' that ↩
could not be found.

The injection point has the following annotations:
 - @org.springframework.beans.factory.annotation.Autowired(required=true)


Action:

Consider defining a bean of type 'java.lang.String' in your configuration.
```

Spring Boot makes use of a *FailureAnalyzer*, which is typical for Spring Boot but unknown to the Spring Framework. In Spring Framework–only applications, only the exception is written to the log stream, not such a detailed description. This error analysis provides a friendly output with an explanation.[9]

The information from the FailureAnalyzer always consists of two parts: a `Description`, that is, an error description, and an action, what we can do to make the error disappear. The description says that a field (i.e., the instance variable) in our class requires something, namely a `String`, but it doesn't exist at all. One line down, `@Autowired` shows an interesting hint, which will be important for us in a moment: `required = true`. Yes, the bean is "required", that is, currently necessary.

In `Action`, a solution is presented. We need to include a `String` as a Spring-managed bean in our configuration so that this component ends up in the container.

Tip

IntelliJ tries to find out if a wiring works. The development environment isn't always right.

2.5.7 Optional Dependency

There are reasons why not all desired components are always available. A typical example is a plug-in system where one customer buys an extension and another customer doesn't. Then some kind of "weak" wiring must be possible so that there is no error when the component is missing. This is what the Spring Framework allows, and such places need to be specially marked.

We'll look at four options in dealing with optional components:

- Annotation attributes
- `@Nullable`
- `Optional`
- `ObjectProvider`

In the following examples, let's assume `Thumbnail` can produce thumbnails of an image, but this component might be missing.

Option 1: Annotation Attribute

```
@Autowired( required = false )
Thumbnail maybeThumbnail;
```

---

[9] It's not that difficult to write a failure analyzer yourself. All you have to do is create a subclass of `AbstractFailureAnalyzer` and link it to the exception you want to document in particular. Finally, the class is referenced in a configuration file.

The annotation `@Autowired` has the annotation attribute `required`, which we can set to `false`; then it's alright if the component is missing. The instance variable remains `null` in that case. We've seen the possibility in the error message from the FailureAnalyzer.

Option 2: @Nullable

Note

This is from any package, for example, *javax.annotation.Nullable* or *org.springframework.lang.Nullable*.

```
@Autowired
@Nullable
Thumbnail maybeThumbnail;
```

A second option is the use of a special `@Nullable` annotation. Java can't express that certain variables can be `null` or never be `null`—that's why developers have created various frameworks for static code analysis. This isn't standardized, but Spring doesn't care about the package in which `@Nullable` resides. In our example, if there is no `Thumbnail` as a Spring-managed bean, then this instance variable also remains `null`.

Option 3: Special Data Type: Optional

```
@Autowired
Optional<Thumbnail> maybeThumbnail;
```

The third option is to use the special data type `Optional`. `Optional` is a container that may contain something or may contain nothing. For optional references, this is a good field of application because the instance variable never remains `null`. The user doesn't need to remember testing for `== null` or `!= null` and won't run into a `NullPointerException` accidentally. As a type for fields, `Optional` is rather uncommon, and serialization isn't possible. And it would also be unusual in the parameter list for a constructor or setter.

`Optional` has the useful methods `orElse(…)` and `orElseGet(…)`, with which we can select an alternative object exactly when there is no component in `Optional`. This functionality also has the next option.

Option 4: ObjectProvider

```
@Autowired
ObjectProvider<Thumbnail> maybeThumbnail;
```

`ObjectProvider` is used similarly as `Optional`. The key difference is that in `Optional`, the reference is already set and resolved when the component tree is built, whereas with

`ObjectProvider`, it's fetched live from the context only when the query is made. In addition, the `ObjectProvider` can handle ambiguities, for example, when there are multiple `Thumbnail` implementations. We'll come back to this in Section 2.7.4 .

Tip

Setters are good for optional dependencies. Things that are absolutely necessary are best enforced via a parameterized constructor.

2.5.8 Cyclic Dependencies *

By default, if you want to implement cyclic dependencies in Spring, it leads to an exception.[10] Let's look at the following:

```
@Component class A { @Autowired B b; }
@Component class B { @Autowired A a; }
```

There is an exception at startup.

Here, `A` wants a reference to `B`, and `B` wants a reference to `A`. This leads to a cross dependency. This isn't allowed by default in field and constructor injection and results in an error reported as follows:

```
The dependencies of some of the beans in the application context form a cycle:

   circularApplication
┌─────┐
|  a (field ... .B ... .A.b)
↑      ↓
|  b (field ... .A ... .B.a)
└─────┘

Action:

Relying upon circular references is discouraged and they are prohibited ↩
by default. Update your application to remove the dependency cycle between ↩
beans. As a last resort, it may be possible to break the cycle ↩
automatically by setting spring.main.allow-circular-references to true.
```

The problem is visualized nicely. We know the representation with the FailureAnalyzer because we've seen something similar in Section 2.5.6.

The FailureAnalyzer shows us a solution, and that is to set configuration property `spring.main.allow-circular-references` to `true`. There are other possible solutions as well. Surprisingly, the scenario with setter injection is possible. This is surprising because the different injection types should ultimately lead to the same goal. The third possibility is provided by the annotation `@Lazy`, which we'll refer to later in Section 2.8.2.

---

[10] Since Spring Boot 2.6, before that was fine: *https://github.com/spring-projects/spring-boot/wiki/Spring-Boot-2.6-Release-Notes*

So, we have three ways to work with circular dependencies. The question is whether you really want that. The message makes the following clear:

```
Relying upon circular references is discouraged and they are prohibited by
default.
```

Normally, these models are design issues, and it's recommended to remove the dependency cycles between these beans. However, migrating an application that was created before Spring Boot 2.6 to a current Spring Boot version may require this hack.

2.5.9 Inject Other Things

In addition to our own components, Spring Boot puts other managed beans in context that we can have injected—at the beginning, when we listed these components, we could also see them.

In addition, there are other components that *aren't* Spring-managed beans, but that you can still have injected. Among them is the context itself:

```
@Autowired
ApplicationContext applicationContext;
```

You can have the context injected into your own code and get beans out of the context this way. But this way is rather wrong. The right way is to wish for something because that is exactly what IoC is: I give away my control, but eventually, I get it anyway. But of course, getting it out explicitly might still be useful, for example, if beans are automatically named after a pattern and should be taken out in your own code.

In addition to `BeanFactory` and `ApplicationContext` (and their concrete subtypes, such as `ConfigurableApplicationContext`), other types can be injected:

- `Environment`: Request configuration information
- `ResourceLoader`: Load resources
- `ApplicationEventPublisher`: Send events
- `MessageSource`: Request translations

This includes the concrete subtypes, such as `ResourcePatternResolver`, which is a subtype of `ResourceLoader` that can also search with wildcards.

We can also have injected into our code `ApplicationArguments`, which are arguments passed on the command line. This can be an alternative to implementing the `[CommandLine|Application]Runner` interfaces, through which we also have access to the command-line options.

2.6 Configuration Classes and Factory Methods

In this section, we'll look at *configuration classes* and *factory methods*. We'll see that besides the `@Component` annotation, there is another annotation and another way to create Spring-managed beans.

2.6.1 What @Component Can't Do

First, let's examine the limitations of beans created using the `@Component` annotation. Spring will call the constructor, either the parameterless or the parameterized constructor, and the only way to customize the instance is through the constructor. So, the initialization is done by the component itself.

A second disadvantage is that it's not possible to build different instances depending on certain states. That is, if a component is annotated with `@Component`, then it's basically included—in principle, control is also possible. (More about this is given later in Section 2.10).

In addition, what do we do with classes that aren't annotated with `@Component` at all? We might want to include them as Spring-managed beans as well.

That's precisely why there is another possibility besides `@Component` types: factory methods. These are custom methods that build and return instances. When we have factory methods, they give us absolute freedom in how we build instances. Then, it's easy to bind the instances to certain conditions on which a subclass `A`, `B`, or `C` can be built depending on the context.

Moreover, we can build arbitrary instances of arbitrary types and then easily put them into context as a Spring-managed bean. Next we'll see what this looks like in practice.

2.6.2 @Configuration and @Bean

With proprietary factory methods, several things come together. Here's a fictitious example:

```
@Configuration
public class MyOwnBeanFactory {

  @Bean
  Foo method_name_will_be_bean_name() { return … }

  @Bean
  Bar another_factory_method() { return … }
}
```

Here, you can see the following:

- The classes are annotated with `@Configuration`—in principle, `@Component` is possible, which we'll talk more about later.
- Factory methods are annotated with `@Bean`. These `@Bean` annotated methods are called automatically, and they return the exact bean that comes into the context as a Spring-managed bean.
- The visibility of the methods should be `public` or package-visible. The class and methods must not be `final` by default.
- The return type should be as precise as possible. This sounds strange at first because countless operations run dynamically at runtime, of course. Actually, it shouldn't matter if the method returns a `java.lang.Object` or a precise type, but in the Spring Framework, certain information isn't evaluated at runtime and is extracted from the bytecode. This precise type information is needed for conditions, for example, and these conditions are evaluated with the static type information and not with the runtime type. Therefore, we should get into the habit of always specifying the return type as precisely as possible.
- The method name is in principle freely selectable, but this will later automatically be the name of the bean.

Spring beans can be injected and taken out of context because how the beans ultimately got into context doesn't really matter.

@Configuration Is an @Component

The annotation type `@Configuration` shows that it has the meta-annotation `@Component` on it:

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Component
public @interface Configuration { … }
```

In other words, anything annotated with `@Configuration` is like an `@Component` and is recognized via classpath scanning.

The `@Configuration` classes can also be nested static classes, which incidentally applies to `@Component` classes as well.

Note

It's not mandatory that `@Bean` methods always reside in an `@Configuration` annotated class. In principle, `@Component` classes can also be defined as containers for factory methods. However, these *lite beans* (lightweight beans) are restricted. We'll look at this in Section 2.6.4 and take a closer look at this bean type.

@SpringBootApplication Is an @Configuration

The annotation type `@SpringBootApplication` is indirectly also an `@Configuration`, more precisely an `@SpringBootConfiguration`, which is then an `@Configuration`:

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@SpringBootConfiguration
@EnableAutoConfiguration
@ComponentScan(…)
public @interface SpringBootApplication {
  …
}
```

That is, in a class annotated with `@SpringBootApplication`, `@Bean` methods are also possible.

To recall our initial setup, we listed all the beans, and the main class had an `@SpringBootApplication` annotation. The main class annotated with `@SpringBootApplication` appeared. Here's why: `@SpringBootApplication` is an `@SpringBootConfiguration`, which in turn is an `@Configuration`, which is an `@Component`, and the initial configuration passed in on `run(…)` takes Spring Boot into context.

PromptProvider Interface

The Spring Shell declares an interface called `PromptProvider` for the string (aka prompt) on the shell. By default, this is the string `shell:>`. The nice thing is that we can also implement our own `PromptProvider` and put it in context. Spring Boot will then honor the custom component and ignore the default implementation. This is part of the magic auto-configuration that we'll look at in more detail later; namely, if you provide your own implementation of certain types, auto-configuration will no longer create a bean. More on this feature of the Spring Framework follows in Section 2.10.

The `PromptProvider` interface is a functional interface with one method (see Figure 2.7).

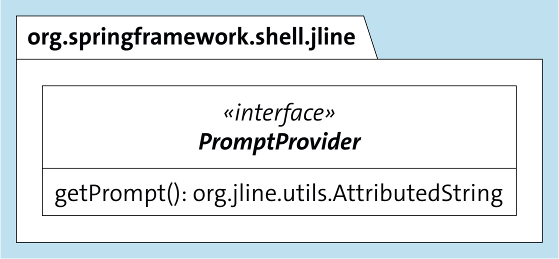

org.springframework.shell.jline
«interface»
PromptProvider
getPrompt(): org.jline.utils.AttributedString

Figure 2.7 "PromptProvider" Interface of the Spring Shell

A custom implementation must implement the `getPrompt()` method. The return is an `AttributedString`. From the package name, you can see that this isn't a specific data type from the Spring universe, but it comes from a command-line library called *JLine*.[11]

PromptProvider as Class

We can take two approaches when a component is required that is of type `PromptProvider` variant and is to write a class that implements `PromptProvider` and overrides the `getPrompt()` method:

```
import org.jline.utils.AttributedString;
import org.springframework.shell.jline.PromptProvider;

@Component
class MyPromptProvider implements PromptProvider {

  @Override
  public AttributedString getPrompt() {
    return new AttributedString( "date4u:>" );
  }
}
```

The `getPrompt()` method returns the `AttributedString`; we build the object via the constructor. The argument is the prompt string.

At one point, the shell will say, "Give me a `PromptProvider`." Because our class and component is of type `PromptProvider`, the shell will use our component for the shell prompt.

Suppose we need to handle different provider requests based on specific runtime conditions. For example, we may have administrators with elevated privileges who require a different string and color for their prompts. In such cases, we can use factory methods to generate different prompts based on the conditions that arise only at runtime.

Return Different PromptProviders Depending on a Condition in an @Bean Method

The methods annotated with `@Bean` should be put into an `@Configuration` class. Because `@SpringBootApplication` is an `@Configuration`, the factory method can be put directly into the main class. In pseudocode, it can look like this:

```
@Configuration
public class PromptProviderConfig {

  private static final PromptProvider userPromptProvider = () ->
      new AttributedString( "date4u:>" );
```

---

[11] *https://jline.github.io*

```
  private static final PromptProvider adminPromptProvider = () ->
      new AttributedString( "date4u[admin]:>",
            AttributedStyle.DEFAULT.foreground( AttributedStyle.RED ) );

  @Bean
  PromptProvider myPromptProvider() {
    if ( is admin  )
      return adminPromptProvider;
    else
      return userPromptProvider;
  }
}
```

The new `@Bean` method is called `myPromptProvider()`, which would give it the same name as from the `@Component` class. The `@Bean` method returns an implementation of `PromptProvider` depending on conditions; the code expresses this in pseudocode. There is a lot of flexibility in being able to independently determine what this instance looks like. This dynamic possibility isn't allowed by `@Component`.

Various Beans Allowed as Types

We've noticed that components can be created and put into the context as a Spring-managed bean in two ways: by `@Component` classes and by factory methods. The interesting thing is that `@Bean` methods don't have to supply their instances of self-written classes at all; they can be any data type you can instantiate. This also includes simple data types, such as a `String` instance or data structures such as arrays or lists:

```
@Bean String name() {
  return "Fillmore";
}

@Bean String[] names() {
  return new String[]{ "fillmore", "juicylucy" };
}

@Bean List<String> namesList() {
  return List.of( "fillmore", "juicylucy" );
}
```

The small example shows three `@Bean` methods. References to these objects can be injected with `@Autowired` just as references to the complex business objects:

```
@Autowired String name;
@Autowired String[] names;
@Autowired List<String> namesList;
```

We can see that an `@Autowired` for a Spring-managed bean of the simple type `String` is just as possible as the array or a generic type such as a list of strings. You only have to be careful with ambiguities, which we'll discuss later.

Note: Question of Understanding

Given the following class:

```
@Configuration    // https://tiny.one/yckrjmxc
class AppUuidConfig {
  private final Logger log = LoggerFactory.getLogger( getClass() );

  @Bean
  UUID appUuid() {
    UUID uuid = UUID.randomUUID();
    log.info( "uuid -> {}", uuid );
    return uuid;
  }
}
```

The idea of this configuration is that the application gets a unique ID of the type `UUID` at startup time.[12] The `UUID` is formed initially and then comes into the context as a Spring-managed bean. The final class `UUID` comes from the Java SE, which means you can't work with an `@Component` annotation on, for example, a subclass, so the program uses a factory method.

This brings up two questions: Will the factory method be called, even if there is no wiring for a Universally Unique Identifier (UUID)? If you inject two UUIDs into your code with `@Autowired`, is the factory method called once or twice?

The answer is that the `appUuid()` method is called only once, so the Spring-managed bean is instantiated only once. Furthermore, it doesn't matter whether a UUID is needed or not. This isn't surprising because it's exactly the same for `@Component` components: whether a component is needed or not is completely irrelevant; every component is built. The framework could also not even determine whether this component is needed later because the component could be fetched from the context at another time. This means `@Bean` methods, just like classes annotated with `@Component`, are automatically called when the application is started, that is, when the container is built, and the entire object graph is initialized. Remember, when the container has been started, you know that all those constructors and factory methods are valid and have been processed correctly.

This basically also answers the questions of what would happen with two `@Autowired`-UUID injections: there is only one instance of UUID. Whether that isn't wired at all or is wired 100 times doesn't change the fact that the UUID object is in context. The factory method is called only once, at least for the normal singleton beans. There is also a prototype bean, which is introduced briefly in Section 2.8.10.

---

[12] A *Universally Unique Identifier* (UUID) is a 128-bit random number.

Same Types Are Allowed as Spring-Managed Beans

Basically, several beans of the same types are allowed in the context, as the following example shows:

```
@Configuration
class AppUuidConfig {
  private final Logger log = LoggerFactory.getLogger( getClass() );

  @Bean String appUuid() {
    String uuid = UUID.randomUUID().toString();
    log.info( "uuid -> {}", uuid );
    return uuid;
  }

  @Bean String secondAppUuid() {
    String uuid = UUID.randomUUID().toString();
    log.info( "uuid -> {}", uuid );
    return uuid;
  }
}
```

The example isn't significantly different from the first example. The only difference is the `secondAppUuid()` method, which also returns a `String`. So multiple Spring-managed beans of the same type aren't a problem. The only problem is the wiring because it's our task to identify the bean unambiguously—this is the subject of a Section 2.7.3, which deals with the basic types and their assignment.

Same @Bean Method Names Aren't Allowed

To extend the program, we'll introduce a new class:

```
@Configuration
class AppUuidConfig { // as before
 @Bean String appUuid() { … }
 @Bean String secondAppUuid() { … }
}
@Configuration
class AppUuidConfig2 {
  @Bean String appUuid() { … }
  @Bean String secondAppUuid() { … }
}
```

It's noticeable that the two classes have the same factory methods. We know that one, two, or four strings are fine, but would the container start the program? The answer is no. The problem is that two factory methods have the same name due to the automatically chosen bean names. Every Spring-managed bean gets a name by default: For classes, it's the lowercase class name, and for `@Bean` methods, it's the method name. If a bean was previously created with the names `appUuid` and `secondAppUuid`, it must not be defined again.

How is the problem solved? In practice, conflicts are surprisingly rare for the reason that method names are often very descriptive and long. That is, a short method such as `appUuid()` is rather uncommon in the Spring environment. With long method names, you can reduce conflicts of a possible collision well. In principle, however, you could also give these beans individual names. To see how this works, Section 2.7.1.

Test Question

We have a configuration, `UuidConfiguration`, which should be detected via classpath scanning as usual. Additionally, there is an `@Bean` method that returns a `String`:

```
@Configuration
public class UuidConfiguration {

  /**
   * @return a random UUID as String.
   */
  @Bean String uuidConfiguration() {
    return UUID.randomUUID().toString();
  }
}
```

Question: Does the program work and does it end up with two Spring-managed beans from `@Configuration` and `@Bean`?

Solution: The way the example is programmed, it won't work because we have a conflict in naming the components. We said that the components get an automatic name. This name is the lowercase name of the class, so `UuidConfiguration` becomes the bean name `uuidConfiguration`. But the factory method is called `uuidConfiguration()`, and that bean would also become `uuidConfiguration`. This leads to a conflict because the bean names must be unique.

There are four ways to solve the problem:

- Change the name of the class.
- Change the name of the method.
- Give this component an explicit name.
- Explicitly give a name to the Spring-managed bean built by the `@Bean` method.

That is, we must intervene in the cases where these methods "happen" to be named like the classes. Otherwise, the container won't start.

2.6.3 Parameter Injection of an @Bean Method

For `@Component` classes, the container calls the constructor of the component. If we look at an `@Bean` method, the container calls it also. There is a connection: the factory method is similar to a called constructor in a component. But with regular components, there are parameterized constructors that can express a wish for something, namely references to other objects or even configuration information.

The same works for factory methods as well—they can also have a parameter list. Here's an example:

```
@Bean
public WatcherService newWatcherService( FileSystem fs ) { … }
```

In this fictitious example, the factory method is to build a `WatcherService`, but the method needs the file system to do so.

There are also optional components, that is, components that don't necessarily need to be present as a Spring-managed bean. We discussed various solutions in Section 2.5.7. For example, an `Optional` can be used:

```
@Bean
public WatcherService newWatcherService( Optional<FileSystem> fs ) { … }
```

The type is unusual for parameters and only common for method returns.

2.6.4 @Configuration Bean and Lite Bean

To illustrate another feature, let's modify the example slightly.

```
@Configuration
class AppUuidConfig {
  private final Logger log = LoggerFactory.getLogger( getClass() );
  public AppUuidConfig() { log.info( getClass().getName() ); }
  @Bean String appUuid() {
    String uuid = UUID.randomUUID().toString();
    log.info( "uuid -> {}", uuid );
    return uuid;
  }
  @Bean String shortAppUuid() {
    String uuid = appUuid().substring( 0, appUuid().length() / 2 );
    log.info( "short uuid -> {}", uuid );
    return uuid;
  }
}
```

In addition to the `appUuid()` method, there is a second method, `shortAppUuid()`, which—and this is the special feature—falls back on `appUuid()`. The `shortAppUuid()` method simply takes half of the original UUID.

The question is: What output can be expected? Without specific prior knowledge, one would assume that `shortAppUuid()` outputs half of a completely new UUID because `shortAppUuid()` relies on `appUuid()` and a `UUID.randomUUID().toString()` is called with each invocation.

The surprising part is that `shortAppUuid()` really prints half of the return of `appUuid()`. For example, the logger would write the following:

```
uuid -> 2723de4b-a53a-460f-92f3-a9243ca65bef
short uuid -> 2723de4b-a53a-460f
```

If the `appUuid()` method is called multiple times, the state computed the first time is always taken, and the method isn't called again. That's actually what factory methods are supposed to do: build the instance, and then Spring puts it into the container as a managed bean and injects it.

More interestingly, how is this possible? It looks like a regular method call. The log output gives a hint because it shows the `Class` object and that doesn't show our class:

```
2033-01-02 10:20:30.834  INFO 108576 --- [ main] ↩
dConfig$$EnhancerBySpringCGLIB$$b25663cd : uuid -> …
2723de4b-a53a-460f-92f3-a9243ca65bef
2033-01-02 10:20:30.835  INFO 108576 --- [ main] ↩
dConfig$$EnhancerBySpringCGLIB$$b25663cd : short uuid -> …
```

The output `EnhancerBySpringCGLIB` makes it clear that you aren't dealing with an original instance, but with a proxy object. Spring builds a subclass of `AppUuidConfig` and overrides these methods. If the methods are called multiple times, it doesn't matter because these overridden methods return what was built before. Via dynamic binding, when we call the `appUuid()` method in `shortAppUuid()`, we end up not with our own class, but with the subclass generated at runtime.

IDE Note

`@Bean` methods in a standard `@Configuration` must not be `final`. The editor would mark the error with red squiggles. If you use final methods, the Spring Framework will throw an exception.

Lite Bean

When you use `@Configuration` classes, Spring automatically creates a proxy object so that your own methods can fall back on other methods. The approach guarantees that only one instance is delivered, no matter how many times you call the method.

Of course, building such a proxy isn't free. There are two approaches to avoid proxy building. One is to annotate the class with `@Component` instead of `@Configuration`. Then no proxy is built anymore. The other method is discussed in the next subsection.

IDE Note

If you call the `@Bean` methods among each other in an `@Component` class, it's probably an error. IntelliJ marks the problem area with red squiggles.

@Configuration(proxyBeanMethods = false)

Marking a class as `@Component` is possible, but it's the wrong signal for configurations. There is another way to disable the proxy:

```
@Configuration( proxyBeanMethods = false )
class AppUuidConfig {
  …
}
```

If `proxyBeanMethods = false` is set, no proxy will be generated anymore. It's also a good standard to always set the flag because the scenario "`@Bean` method calls other `@Bean` method" doesn't happen that often. If the goal is native compilation, this is a good practice anyway because the less magic is used at runtime, the easier native compilation is.

IDE Note

If you have `@Configuration(proxyBeanMethods = false)`, and yet you call the methods among each other, IntelliJ will mark these places as errors. This is a good hint, so you can basically start with `proxyBeanMethods = false`, and if you do call `@Bean` methods among each other, you can remove `proxyBeanMethods = false` or rewrite the code.

2.6.5 InjectionPoint *

Factory methods, like constructors, can request references to other Spring-managed beans. The factory method can work with these referenced beans.

A factory method may also request information about the injection point via the parameter list. The factory method may want to build or configure different Spring-managed beans depending on the injection point. It's possible that if a bean is requested by class `A`, it should be built differently than if it were requested by class `B`. This is precisely why you may want this special type `InjectionPoint`.

Here's a fictitious example: The `@Bean` method `random()` will return a `Random` object, but depending on the injection point, `Random` objects of different quality will be returned:

```
@Bean Random random( InjectionPoint injectionPoint )
```

Interface `InjectionPoint`[13] originates from the Spring Framework. `InjectionPoint` provides different information, which mainly comes from the `java.lang.reflect` package (see Figure 2.8).

Under `java.lang.reflect`, we have, among other things, data type `Field`, which is a `Member` and an `AccessibleObject`. An `AccessibleObject` is an `AnnotatedElement`. `Annotation` is a data type from `java.lang.annotation`.

The methods of `InjectionPoint` return these types when retrieving the metadata. This can be seen well in `getField()`, `getMember()`, and `getAnnotations()`. Spring declares its own data type for method parameters in `org.springframework.core`, and this is also returned in a method at `InjectionPoint`.

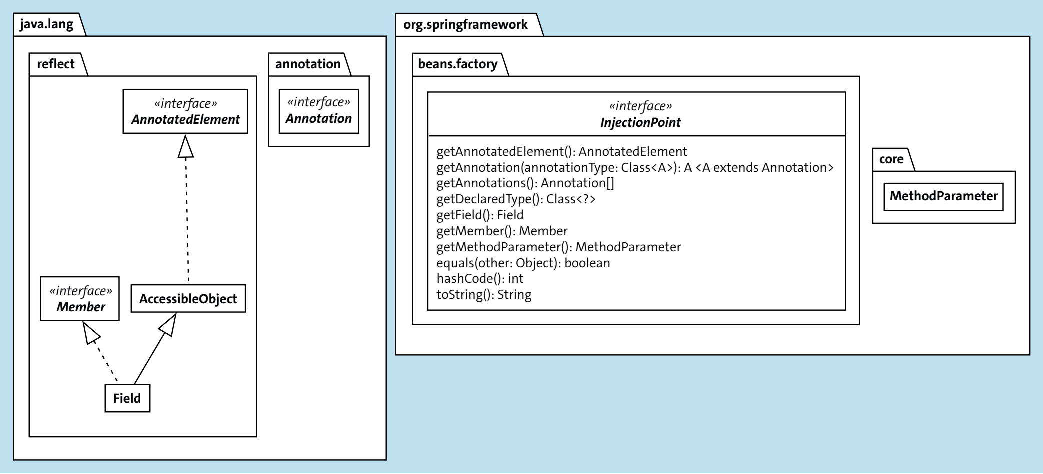

java.lang
reflect
«interface»
AnnotatedElement
«interface»
Member
AccessibleObject
Field
annotation
«interface»
Annotation
org.springframework
beans.factory
«interface»
InjectionPoint
getAnnotatedElement(): AnnotatedElement
getAnnotation(annotationType: Class<A>): A <A extends Annotation>
getAnnotations(): Annotation[]
getDeclaredType(): Class<?>
getField(): Field
getMember(): Member
getMethodParameter(): MethodParameter
equals(other: Object): boolean
hashCode(): int
toString(): String
core
MethodParameter

Figure 2.8 Injection Point and Reflection Types

Let's take a look at a complete example: The factory method `random()` returns `Random` objects, but if the injection point is annotated with `@CryptographicallyStrong`, it should not return a normal `java.util.Random` object, but a better `SecureRandom`, which is a subtype of `Random`. Because `@CryptographicallyStrong` isn't a framework annotation, we declare the following:

```
@Retention( RetentionPolicy.RUNTIME )
@interface CryptographicallyStrong { }
```

The annotation type must be evaluated at runtime, of course, so `RetentionPolicy.RUNTIME` is set.

Let's look at three examples of how we can use annotation:

---

[13] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/beans/factory/InjectionPoint.html*

```
@Autowired
@CryptographicallyStrong Random random;

@Autowired
@CryptographicallyStrong
void setRnd( Random random ) { … }

@Autowired
void setRnd( @CryptographicallyStrong Random random ) { … }
```

We want to use the annotation for field injection and setter injection. In the case of setter injection, the annotation has to be evaluated at the injection point and at the method; this makes a small difference in the programming.

With the factory method, we can fall back on `InjectionPoint`. This allows the program to evaluate whether the injection target is annotated with `@CryptographicallyStrong`.

```
@Bean @Scope( ConfigurableBeanFactory.SCOPE_PROTOTYPE )
public Random random( InjectionPoint injectionPoint ) {
  return nonNull(injectionPoint.getAnnotation(CryptographicallyStrong.class))
         || (  injectionPoint.getMember() instanceof AnnotatedElement member
           && member.isAnnotationPresent( CryptographicallyStrong.class ))
         ? new SecureRandom()
         : new Random();
}
```

The `@Bean` method `random()` "wants" information about the injection point from the Spring Framework. First, the method asks if the annotation from the injection target is `@CryptographicallyStrong`. If `injectionPoint.getAnnotation(CryptographicallyStrong.class)` isn't `null`, the annotation is set—the method `nonNull(…)` is statically imported from the `Objects` class.

This test isn't necessarily sufficient because we would like the setter method to be annotated as well. For this case, an additional test must be programmed. If the `member` can in principle carry annotations (i.e., `instanceof AnnotatedElement` returns `true`), then the pattern variable `member` is initialized. A second test follows whether the annotation `@CryptographicallyStrong` is set on this `AnnotatedMember`.

If we find annotation `@CryptographicallyStrong` on the instance variable, parameter variable, or setter, the method returns `SecureRandom`; otherwise, it returns `java.util.Random`.

There is one important feature to consider. The instances returned from the `@Bean` methods, like those instances from `@Component` components, are singletons by default. That is, they are instantiated only once and then go into the context object; from there, they are injected crisscross to all the places that—in our case—want a `Random`. A normal `@Bean` implementation would only build a `Random` object once, and then rather "randomly" the first injection point would determine whether it becomes a `Random` object or a `SecureRandom` object. It's probably not intentional that the first random injection point determines whether `Random` or `SecureRandom` objects are later distributed throughout the framework. We probably want different `Random` and `SecureRandom` objects depending on whether the annotation is set or not.

In addition to singletons, the Spring Framework can also always rebuild an object depending on the requirements, which brings up *prototypes*. Whether a Spring bean is a singleton or a prototype is determined by the annotation `@Scope`:

```
@Bean
@Scope( ConfigurableBeanFactory.SCOPE_PROTOTYPE )
public Random random( InjectionPoint injectionPoint )
```

The line sets the prototype scope for `Random` objects. Thus, our program always decides anew whether a `Random` or a `SecureRandom` object should be returned due to the existence of `@CryptographicallyStrong`.

Prototype beans don't play a particularly large role in this book, but this is one of the few places where these prototypes make sense. Section 2.8.10, presents the difference in a bit more detail.

2.6.6 Static @Bean Methods *

The methods annotated with `@Bean` can be static in principle. The advantage is that for a static method, no instance of the `@Configuration` class must exist. And that's also the point of these static `@Bean` methods because they provide Spring-managed beans very early in the application's lifecycle, temporally before the surrounding `@Configuration` bean. Of course, mutual `@Bean` calls don't work then either because that only works by Spring generating a subclass and then, dynamically bound, calling a subclass method. Static methods, however, are never dynamically bound in Java as a matter of principle. That is, if a static `@Bean` method calls another static `@Bean` method, then we always stay in the superclass and don't go into the subclass dynamically bound.

Static `@Bean` methods may be necessary when a bean is needed very early in the lifecycle, regardless of the surrounding configuration. An example can be found in the JavaDoc of `@Bean`[14] and in the reference documentation of the Spring Framework: *https://docs.spring.io/spring-framework/docs/current/reference/html/core.html#beans-factorybeans-annotations*.

---

[14] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/context/annotation/Bean.html*

2.6.7 @Import and @ImportSelector *

We've seen the basic properties for configuration and `@Component` classes, and now we'll look at a few special cases.

@Import

We'll begin by introducing three classes:

```
@Component
class Water {}

@Configuration
class CoffeeBeansConfig {
  @Bean public CoffeeBeans beans() { … }
}

@Configuration
@Import( { Water.class, CoffeeBeansConfig.class } )
class CoffeeConfig {
  @Bean public Coffee coffee( CoffeeBeans bean ) { … }
}
```

The `Water` class becomes a simple component, and `CoffeeBean` is created via a factory method. An `@Autowired` on `Water` or `CoffeeBeans` is possible.

The `@Import` annotations can be used to include types, and it's nice to do this with a configuration so that one configuration includes other configurations. For example, `CoffeeConfig` introduces a third Spring-managed bean `Coffee` via a factory method.

Now, let's focus on the line `@Import( { Water.class, CoffeeBeansConfig.class } )`. The annotation imports the other types `Water` and `CoffeeBeansConfig`. The question is whether this `@Import` annotation is necessary or can be omitted. The answer is that, of course, `@Import` can be omitted because `Water` and `CoffeeBeansConfig` are recognized via classpath scanning. So, what can `@Import` be used for?

Two use cases for the `@Import` annotation are as follows:

- It can also be used to include configurations that are in arbitrary packages. Normally, classpath scanning recognizes specific base packages, but `@Import` can be used to import anything, including configurations and components that can be in any packages outside the scanned packages.
- With `@Import`, you can abandon classpath scanning completely and thus save the costs for the search. `@Import` would then only have to explicitly enumerate all classes and factories.

Tip

It’s recommended to use `@Import` so that a main configuration can reference subconfigurations. For example, these subconfigurations configure beans for the infrastructure, such as messaging system, database system, email system, and so on.

@ImportSelector

The specification via `@Import` is always static, and the type is noted in the code. Besides `@Import`, Spring Framework supports an `@ImportSelector` that can decide at runtime what should be included or not. We can see how something like this works with this small example:

```
class CoffeeBeansImportSelector implements ImportSelector {
  @Override public String[] selectImports( AnnotationMetadata __ ) {
    if ( Math.random() > 0.5 )
      return new String[]{ JavaCoffeeBeansConfig.class.getName() };
    else
      return new String[]{ BrazilianCoffeeBeansConfig.class.getName() };
  }
}
```

The class implements functional interface `ImportSelector` with method `selectImports(…)`. The method gets parameter `AnnotationMetadata`, which our example program doesn’t need and which is therefore commented out with two underscores. This method returns an array of fully qualified class names, where the class names are specified as strings, not as `Class` objects. Randomly, one configuration or another is returned. To summarize, `@ImportSelector` contains logic that returns the fully qualified types for `@Configuration` at runtime. Here it happens to be `JavaCoffeeBeansConfig` or `BrazilianCoffeeBeansConfig`.

If you want to use such a selector, specify it in `@Import`:

```
@Configuration
@Import( CoffeeBeansImportSelector.class )
static class CoffeeDrinker { }
```

The logic is a bit comparable to factory methods because they also decide at runtime what the instance should look like. With `@Import`, `ImportSelector` also decides at runtime which types to include.

A test could look like this in practice: If the switch is set to database 1, then configure the database driver for this database 1. Or is it database 2? Then we initialize the configuration for database 2, and so on.

2.7 Abstraction and Qualifications

We've noticed that there are several ways to build Spring-managed beans. We can define components via `@Component` and with the semantic annotations such as `@Repository`, `@Service`, or `@Controller`, and we can have them built via `@Bean` methods. Ultimately, each Spring component ends up in the big bag with all the other Spring-managed beans. Then we can inject these Spring-managed beans into our code using `@Autowired`.

In this section, we'll look at abstractions and qualifications. With abstractions, you don't need to know the precise type, and this isn't uncommon in Java: we call it "programming against interfaces" in that the actual object type remains hidden. Something similar can also be implemented in the Spring Framework. We can express "I want a bean of type `A`," and I get a concrete implementation or subclass of type `A`, without then knowing exactly what I get.

2.7.1 Bean Name and Alias

One way to introduce an abstraction is by name. Every Spring-managed bean automatically has a name and can even have more than one name. Other optional names are called *aliases*. Without explicit naming, the name is automatically set by the framework: for components, it's the lowercase name of the component, and, for factory methods, it's the name of the factory method.

If the name is known, the bean can be retrieved from the context under this name, so the actual class name is then no longer important.

The name of a Spring-managed bean can be changed. There is a difference between `@Component` components and `@Bean` methods—let's look at both.

Names at @Component

The annotation type `@Component`[15] has exactly one annotation attribute, `value`, through which we can determine the name of the component:

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Indexed
public @interface Component {
  String value() default "";
}
```

By default, no name is assigned, so Spring Framework assigns the name automatically.

---

[15] *http://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/stereotype/Component.html*

Names at @Bean

The annotation type `@Bean` also declares an annotation attribute `value` for an explicit name:

```
public @interface Bean {

  @AliasFor(value="name")
  String[] value() default {};

  @AliasFor(value="value")
  String[] name() default {};

  // …
}
```

However, there are other annotation attributes, and therefore we assign an alternative name for `value`, namely `name`. With multiple annotation attributes, `name` is more descriptive than `value`; the `value` identifier is far too generic to unambiguously indicate what `value` sets for a value. In this context, we would then use `name` when setting more than one annotation attribute at `@Bean`. If only a name is set, we have a compact notation via `value`, which can then be omitted. Two annotation attributes for the same thing occur frequently and are documented with `@AliasFor` documented.

2.7.2 AwtBicubicThumbnail for Thumbnails

We want to add a new service called `AwtBicubicThumbnail` to the application that creates small thumbnails. The built-in Java SE classes from the AWT are helpful in this.

```
@Service
public class AwtBicubicThumbnail {
  private static BufferedImage create( BufferedImage source,
                                       int width, int height ) {
    double thumbRatio = (double) width / height;
    double imageRatio = (double) source.getWidth() / source.getHeight();
    if ( thumbRatio < imageRatio ) height = (int) (width / imageRatio);
    else width = (int) (height * imageRatio);
    BufferedImage thumb = new BufferedImage( width, height,
                                             BufferedImage.TYPE_INT_RGB );
    Graphics2D g2 = thumb.createGraphics();
    g2.setRenderingHint( RenderingHints.KEY_INTERPOLATION,
                         RenderingHints.VALUE_INTERPOLATION_BICUBIC );
    g2.drawImage( source, 0, 0, width, height, null );
    g2.dispose();
    return thumb;
  }
  public byte[] thumbnail( byte[] imageBytes ) {
    try ( InputStream is = new ByteArrayInputStream( imageBytes );
          ByteArrayOutputStream baos = new ByteArrayOutputStream() ) {
      BufferedImage thumbnail = create( ImageIO.read( is ), 200, 200 );
      ImageIO.write( thumbnail, "jpg", baos );
      return baos.toByteArray();
    }
    catch ( IOException e ) {
      throw new UncheckedIOException( e );
    }
  }
}
```

Listing 2.12 AwtBicubicThumbnail.java

One method is public: `byte[] thumbnail(byte[])`. An image is passed in as a byte array and comes out as a thumbnail. The image is output in JPEG format.

Save Photo with Thumbnail

We want to solve a task in which the `PhotoService` interacts with the `FileSystem` and the `AwtBicubicThumbnail` class. Consequently, there are two dependencies, as Figure 2.9 shows.

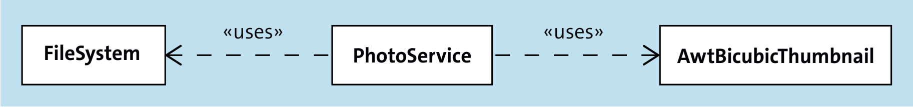


Figure 2.9 "PhotoService" Requiring "FileSystem" and "AwtBicubicThumbnail" Injection

In code, the constructor would have to be adjusted because it's one more wish from the framework.

```
@Service
public class PhotoService {

  private final FileSystem fs;
  private final AwtBicubicThumbnail thumbnail;

  public PhotoService( FileSystem fs, AwtBicubicThumbnail thumbnail ) {
    this.fs = fs;
    this.thumbnail = thumbnail;
  }

  public Optional<byte[]> download( String name ) …
}
```

Listing 2.13 PhotoService.java Extension

For reading images, `PhotoService` already had corresponding method `download(…)`, which relies on the file system in the background. Now, `PhotoService` will also work together with `AwtBicubicThumbnail` and calculate a preview image. This thumbnail is to be used in the save method of an image. The implementation in `PhotoService` looks like the following listing.

```
public String upload( byte[] imageBytes ) {
  String imageName = UUID.randomUUID().toString();
  // First: store original image
  fs.store( imageName + ".jpg", imageBytes );
  // Second: store thumbnail
  byte[] thumbnailBytes = thumbnail.thumbnail( imageBytes );
  fs.store( imageName + "-thumb.jpg", thumbnailBytes );
  return imageName;
}
```

Listing 2.14 PhotoService.java, extension

The `upload(…)` method gets a byte array of exactly the image that is to be saved. This image is stored in two variants: (1) the original image, and (2) the preview image calculated from this byte array and stored as well. Because these images don't have file names, the file name is generated from a UUID. The preview image is automatically given a file extension, so given an image name with the extension `-thumb.jpg`, you could load the preview image. Because this `filename` is generated dynamically, the image name is returned at the end.

`PhotoCommands` already had method `showPhoto(…)` for the metadata. We can now introduce a new shell command for uploading images (see Figure 2.10).

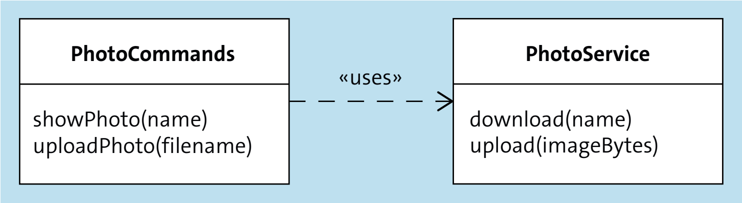


Figure 2.10 "PhotoCommands" Using "PhotoService" to Load/Save Images

The code is shown in the following listing.

```
@ShellMethod( "Upload photo" ) // upload-photo
String uploadPhoto( String filename ) throws IOException {
  byte[] bytes = Files.readAllBytes( Paths.get( filename ) );
  return "Uploaded " + photoService.upload( bytes );
}
```

Listing 2.15 PhotoCommands.java Extension

This new command can then be used as follows:

```
shell:>upload-photo 'C:/Users/User/Desktop/new-image.jpg'
Uploaded 1ab1e0b0-970b-46f0-8bd7-b4b586f01347
```

In the file system, we can then discover two new images, as shown in Figure 2.11.

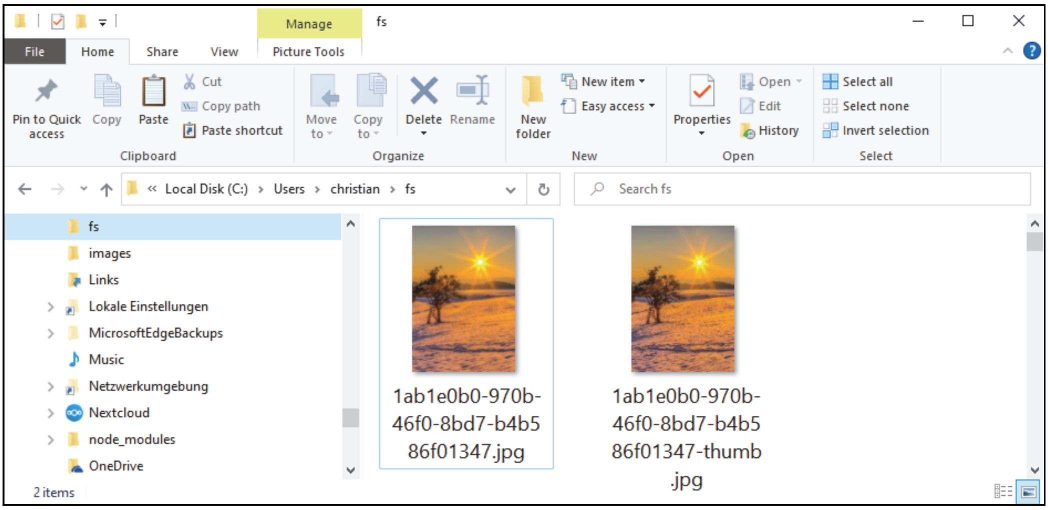

Manage
fs
File
Home
Share
View
Picture Tools
Pin to Quick access
Copy
Paste
Cut
Copy path
Paste shortcut
Clipboard
Move to
Copy to
Delete
Rename
Organize
New folder
New item
Easy access
New
Properties
Open
Edit
History
Open
Select all
Select none
Invert selection
Select
« Local Disk (C:) › Users › christian › fs
Search fs
fs
images
Links
Lokale Einstellungen
MicrosoftEdgeBackups
Music
Netzwerkumgebung
Nextcloud
node_modules
OneDrive
2 items
1ab1e0b0-970b-46f0-8bd7-b4b586f01347.jpg
1ab1e0b0-970b-46f0-8bd7-b4b586f01347-thumb.jpg

Figure 2.11 Example of Uploaded Images

That is, our program does exactly what it should do: we've successfully uploaded an image.

2.7.3 Basic Types

We've shown that you can name components and abstract away from the concrete type via the name. However, you can also abstract away from the concrete type via a supertype. To do this, we introduce a `Thumbnail` interface with a single `thumbnail(…)` method.

```java
public interface Thumbnail {
  byte[] thumbnail( byte[] imageBytes );
}
```

Listing 2.16 Thumbnail.java

The `AwtBicubicThumbnail` class implements the new interface.

```java
@Service
public class AwtBicubicThumbnail implements Thumbnail {
  private static BufferedImage create( BufferedImage source,
                                       int width, int height ) {
```

```
  // as before
  }
  @Override
  public byte[] thumbnail( byte[] imageBytes ) {
  // as before
  }
}
```

Listing 2.17 AwtBicubicThumbnail.java Extension

To use `@Autowired`, it's not necessary to name the exact type `AwtBicubicThumbnail`, but the base type is sufficient so that the following is possible when wiring.

```
@Service
public class PhotoService {
  private final FileSystem fs;
  private final Thumbnail thumbnail;
  public PhotoService( FileSystem fs, Thumbnail thumbnail ) {
    this.fs = fs;
       this.thumbnail = thumbnail;
  }
  …
}
```

Listing 2.18 PhotoService.java Extension

If a program needs something of type `Thumbnail`, the Spring Framework provides the concrete type during injection. We don't even need to know what this object type is called exactly.

Injection Based on the Type *

The Spring Framework is so powerful that it also evaluates generic type information whenever `@Autowired` is used. The following code gives an example:

```
@Configuration
class SupplierConfiguration {
  @Bean Supplier<FileSystem> filesystem() { … }
}

@Component
class PhotoServiceSupplier implements Supplier<PhotoService>{
  @Override public PhotoService get() { … }
}
```

Because we have a `Supplier` for a `FileSystem` and a `Supplier` for a `PhotoService`, that means, strictly speaking, it's only one `Supplier`.

Nevertheless, the Spring Framework evaluates the type information, so the following is possible:

```
@Autowired Supplier<FileSystem> fs;
```

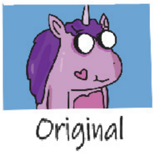
Sibling of AwtBicubicThumbnail: AwtNearestNeighborThumbnail

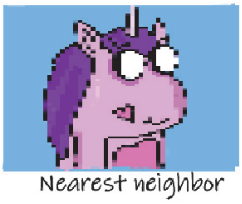
We want to add a second `Thumbnail` implementation that uses a different rendering hint because instead of relying on high-quality scaling, we can simply consider the neighboring pixels, and it's much faster.

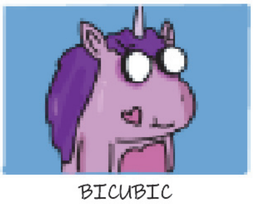
```
@Service
public class AwtNearestNeighborThumbnail implements Thumbnail {
  private static BufferedImage create( BufferedImage source,
                                       int width, int height ) {
    double thumbRatio = (double) width / height;
    double imageRatio = (double) source.getWidth() / source.getHeight();
    if ( thumbRatio < imageRatio ) height = (int) (width / imageRatio);
    else width = (int) (height * imageRatio);
    BufferedImage thumb = new BufferedImage( width, height,
                                             BufferedImage.TYPE_INT_RGB );
    Graphics2D g2 = thumb.createGraphics();
    g2.setRenderingHint( RenderingHints.KEY_INTERPOLATION,
                         RenderingHints.VALUE_INTERPOLATION_NEAREST_NEIGHBOR);
    g2.drawImage( source, 0, 0, width, height, null );
    g2.dispose();
    return thumb;
  }
  @Override
  public byte[] thumbnail( byte[] imageBytes ) {
    try ( InputStream is = new ByteArrayInputStream( imageBytes );
          ByteArrayOutputStream baos = new ByteArrayOutputStream() ) {
      BufferedImage thumbnail = create( ImageIO.read( is ), 200, 200 );
      ImageIO.write( thumbnail, "jpg", baos );
      return baos.toByteArray();
    }
    catch ( IOException e ) {
      throw new UncheckedIOException( e );
    }
  }
}
```

Listing 2.19 AwtNearestNeighborThumbnail.java

The second implementation contains some code duplication, which should be addressed in production code, but we can ignore that. With `RenderingHints.VALUE_INTERPOLATION_NEAREST_NEIGHBOR`, when scaling up, the nearest neighbor is simply placed *n* times next to each other, which of course gives a very pixelated magnification. When images are scaled down, image information is missing. The advantage is that this is a fast-scaling algorithm.







No Unique Bean

With the two classes, the two implementations of `Thumbnail` are now

```
@Service
class AwtBicubicThumbnail implements Thumbnail
```

and

```
@Service
class AwtNearestNeighborThumbnail implements Thumbnail
```

So, we have two candidates that in principle would match a requested type `Thumbnail`. If an injection point expects an instance of the type `Thumbnail`, what will the reaction be?

```
@Autowired Thumbnail thumbnail;
```

There is an ambiguity, and the Spring Framework doesn't pick just any component. The ambiguity is a real error and causes a `NoUniqueBeanDefinitionException` exception at runtime. In other words, whenever we get an error message like this, there are multiple possible candidates for an injection point that need to be resolved.

We have different proposed solutions, and next we'll look at five different ways to solve the problem.

Solution 1: @Autowired with @Qualifier

The first option is to work with the names of the components. We've seen that components are automatically named, and the name can be used to make a selection later. Because we want to get away from concrete type names, it makes sense to rename the components so that `AwtBicubicThumbnail` isn't called `awtBicubicThumbnail` because that would strongly resemble the class name.

I would like to move away from this specific name and call the component `qualityThumbnailRenderer`. That means “a high quality in thumbnail generation.” `AwtNearestNeighborThumbnail`, on the other hand, implements a fast algorithm, that's why the component is called `fastThumbnailRenderer`. The two components are

```
@Service( "qualityThumbnailRenderer" )
class AwtBicubicThumbnail implements Thumbnail …
```

and

```
@Service( "fastThumbnailRenderer" )
class AwtNearestNeighborThumbnail implements Thumbnail …
```

[+]

Tip

Avoid names that are too short because they can quickly lead to collisions. If we were to name the beans `fast` and `slow`, that would be too short because other beans with algorithms can also make something fast or slow.

If the component name is known, we can use `@Autowired` with an additional annotation `@Qualifier` and specify the name of the requested component:

```
@Autowired @Qualifier( "qualityThumbnailRenderer" )
Thumbnail thumbnail;
```

Here, the injection point calls for high-quality `Thumbnail` scaling. So, the name abstracts away from the implementation.

In general, `@Qualifier` can also be used with the other injection types, that is, with constructor injection or setter injection.

This proposed solution is acceptable, as the name is a promising way to abstract away from type names. However, an unpleasant problem remains—misspellings. As a solution, we could introduce constants, but strings remain, and for better type safety, the `@Qualifier` can be used differently.

Solution 2: Custom Annotation Types

`@Qualifier` can be used as a meta-annotation on custom annotation types. There are two different directions that can be used to solve the problem:

- One way is that we write two special annotations, for example, `@ThumbnailRenderingFast` or `@ThumbnailRenderingQuality`, and then pin these annotations to the implementation, for example, at `AwtBicubicThumbnail`.
- A second option is *not to* write different individual annotations, but to write *one* annotation, for example, `@ThumbnailRendering`, and use parameters, such as `FAST` or `QUALITY`. This could be an enumeration type, for example.

We can now examine a second approach:

```
@Target( { ElementType.FIELD, ElementType.METHOD, ElementType.TYPE,
           ElementType.PARAMETER, ElementType.ANNOTATION_TYPE } )
@Retention( RetentionPolicy.RUNTIME )
@Qualifier
public @interface ThumbnailRendering {
  enum RenderingQuality { FAST, QUALITY }
  RenderingQuality value();
}
```

In the first step, we need to declare our own annotation type, and, in the example, this is `@ThumbnailRendering`. This type is itself `@Qualifier`, which is a meta-annotation. In addition, we determine where this annotation type may be attached and that the type must be evaluated at runtime.

Annotation type `@ThumbnailRendering` has an annotation attribute with the default name `value`, and here we can set a `RenderingQuality`. This isn't a string, but an enumeration type, so you can't make a mistake.

Next, the actual implementation, for example, `AwtNearestNeighborThumbnail`, needs to be annotated with `@ThumbnailRendering`:

```
@Service
@ThumbnailRendering( ThumbnailRendering.RenderingQuality.FAST )
class AwtNearestNeighborThumbnail implements Thumbnail
```

We could have set a default in the annotation type, but a `value` is needed here, and because `NearestNeighbor` is fast, the program sets the `RenderingQuality` to `FAST`.

If a concrete implementation is desired later, `@ThumbnailRendering` is used again as a special `@Qualifier`; it determines which component should be injected:

```
@Autowired
@ThumbnailRendering( ThumbnailRendering.RenderingQuality.FAST )
Thumbnail thumbnail;
```

In the example, `ThumbnailRendering` desires a fast implementation.

Solution 3: Primary

Proposed solution number 3 was given by the FailureAnalyzer, and that is the use of the annotation `@Primary`. If there are basically several implementations, `@Primary` can be used to select the main implementation as

```
@Service
@Primary
class AwtBicubicThumbnail implements Thumbnail
```

and

```
@Service
class AwtNearestNeighborThumbnail implements Thumbnail
```

If an injection point requests a `Thumbnail`, the ambiguity is resolved by Spring injecting the component annotated with `@Primary`. The other components are still available in the framework and could be addressed by their name, for example. `@Primary` becomes problematic if several components want to be the primary component because this isn't possible.

Solution 4: All Beans in One Data Structure

This brings us to suggested solution number 4, dealing with multiple components of a type. It's possible to work without `@Qualifier` and say, "I don't want to have the *one* thumbnail implementation; I want to have *all* `Thumbnail` implementations." To achieve this, the instance variable is changed to a data structure:

```
@Service
class Whatever {
@Autowired List<Thumbnail> thumbnails;
// or Set<Thumbnail> thumbnails;
// or Thumbnail[] thumbnails;
}
```

A data structure can hold multiple references, and lists, sets, or arrays are allowed as types. An `@Qualifier` can also be used; that is, given a list, the program might want all `FAST` implementations, for example.

This is a good way to implement a plug-in concept. For example, if there are several plug-ins that transform images, these plug-ins could have a name and appear in the GUI and be selected.

Note

If the `@Autowired` annotation is used with data structures, at least one component must exist. That is, the list will never be empty and must contain at least one element. If the data structure should be able to be empty, we use `@Autowired(required = false)` again.

Besides the possibility to collect something in lists/sets/arrays, there another possibility is `java.util.Map`:

```
@Autowired
// @Qualifier possible
Map<String, Thumbnail> thumbnails;
```

The keys of the map are the component names, and the associated values are the Spring-managed beans. In addition, `@Qualifiers` are possible for restriction.

Solution 5: Go by Name

There is another solution, and it's very magical. Taking `@Qualifier` away for a moment, we get

```
@Service
class AwtBicubicThumbnail implements Thumbnail
```

and

```
@Service
class AwtNearestNeighborThumbnail implements Thumbnail
```

We have two services, and the bean name is the lowercase class name. So `AwtBicubicThumbnail` will be `awtBicubicThumbnail`.

It's possible to choose a variable name that is *identical* to the bean name when wiring:

```
@Autowired Thumbnail awtBicubicThumbnail;
```

That is, in this example, we're looking for a component called `awtBicubicThumbnail`, and our Spring-managed bean is `AwtBicubicThumbnail`. The bean name is the qualifier. This also works with constructor or setter injections.

This strong dependency is madness, and such a thing should be taboo in the program because with a simple refactoring from the class name or variable name, the whole program doesn't work anymore. In addition, the name of the implementation should not be known, as we want to abstract away from it.

Even if this technique works in principle, it belongs to the esoteric Spring features. The Spring documentation calls this technique *fallback match* and strongly advises against it.[16]

@Resource *

With `@Autowired`, the focus is on the type and not the naming—let's disregard `@Qualifier`. In principle, Spring would also search for components by name, but this is rather a special case.

There is another annotation called `@Resource` (*https://jakarta.ee/specifications/annotations/2.0/apidocs/jakarta.annotation/jakarta/annotation/resource*) that deliberately chooses the name of the component as a criterion. Annotation `@Resource` is an alternative to the combination of `@Autowired` and `@Qualifier` with a set name, and it originates from the Java Specification Request (JSR) 250, “Common Annotations for the Java Platform.” In this JSR, other annotation types are also declared, for example `@PostConstruct` and `@PreDestroy`. We’ll refer to these two annotations in the context of Section 2.8.

---

[16] *https://docs.spring.io/spring-framework/docs/current/reference/html/core.html#beans-autowired-annotation-qualifiers*

If you use the `@Resource` annotation, then you’ll only be allowed to use it on instance variables or setters—this annotation isn’t allowed on constructors. Setters may only be JavaBean setters, that is, setters with only one parameter, not what Spring supports by default (i.e., methods with multiple parameters).

Consider the following two components as an example:

```
@Component( "profile" )        class UnicornProfile { }
@Component( "unicornProfile" ) class Profile { }
```

This `UnicornProfile` would by default have the lowercase name of the class, that is, `unicornProfile`, but this component name is changed by the program so that this component is called `profile`. A second component of the `Profile` class type has the bean name `unicornProfile`.

In all cases, `UnicornProfile` is injected:

```
@Component
public class ResourceApplication {
  @Resource( name = "profile" ) void setABC( Object profile ) { }
  @Resource( name = "profile" ) Object abc;
  @Resource                     void setProfile( Object profile ) { }
  @Resource                     Object profile;
}
```

In the example, `@Autowired` isn’t used, but only `@Resource`. The peculiarity is that the name of the component is used, and the type is completely ignored. In addition, we can see that no information for the wiring is given at the `setABC(Object profile)` method and that it’s irrelevant with `@Resource` whether its `ABC` or `Object`. The only relevant part is the resource with the name `profile`, which is to be passed to the setter, and the component with the name `profile` is, in our case, `UnicornProfile`.

The same happens in the second case with a field injection. Again, `Object` and `abc` are completely detached from any types, and the `@Resource` named `profile` references `UnicornProfile` again. So, our code gets injected by the framework `UnicornProfile`.

If we omit the name of `@Resource`, the name is automatically taken from the property of the setter or, at this point, from the name of the instance variable. That means, we can use `@Resource` also without a name.

The Spring team writes in the reference documentation that you should prefer the `@Resource` annotation if you want to select components by name. An alternative to `@Resource` is of course to use the annotation `@Autowired` with `@Qualifier` and a name. Therefore, it’s a matter of taste what developers want to use.

2.7.4 ObjectProvider

In Section 2.5.7, you learned about the `ObjectProvider` data type. Let's assume that it's unclear whether there is a `Thumbnail` bean. Then we can write the following:

```
@Autowired
ObjectProvider<Thumbnail> maybeThumbnail;
```

The `ObjectProvider`[17] is a bit like a `Supplier`, which, when asked for a reference, goes to the context and asks for the bean(s). However, while a `Supplier` only has a `get()` method, `ObjectProvider` has a number of other methods besides the comparable `getObject()` method that look familiar to us from `Optional`, for example, to select alternatives. Let's take a look at `getIfAvailable(…)`:

```
maybeThumbnail.getIfAvailable() // null or bean
maybeThumbnail.getIfAvailable( NoopThumbnail::new )
```

The `getIfAvailable()` method returns either the bean or `null`. In the overloaded variant, we can specify a `Supplier`, for example, through constructor reference. If there is no `Thumbnail`, we can specify the alternative ourselves.

An `ObjectProvider` can be injected into our own program or fetched from the context with a special method called `getBeanProvider(…)`:

```
ObjectProvider<Thumbnail> maybeThumbnail =
    ctx.getBeanProvider( Thumbnail.class );
```

Tip

It's useful to look into the JavaDoc for interface `BeanFactory` (*https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/beans/factory/BeanFactory.html*) because the documentation at `ObjectProvider` is unfortunately not that precise.

ObjectProvider Methods

Let's go through the important methods for `ObjectProvider` (see Figure 2.12).

`ObjectProvider` inherits from `ObjectFactory`, which only declares a `getObject()` method. Because the interface originates from Spring Framework 1.0, `Supplier` wasn't yet invented. In addition, `Supplier` doesn't throw an exception, but `getObject()` can throw a `BeansException`. When the method is called, it looks up the bean in the application context, and three outputs are possible:

---

[17] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/beans/factory/ObjectProvider.html*

- If the method finds exactly one instance, this instance is returned.
- If the method doesn't find an instance, a `NoSuchBeanDefinitionException` is thrown.
- If there is more than one implementation matching this type, this results in a `NoUniqueBeanDefinitionException`.

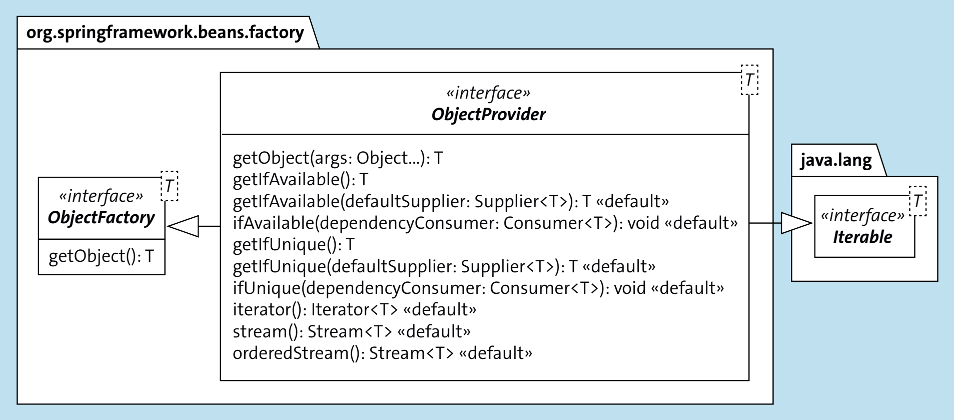

org.springframework.beans.factory
«interface»
ObjectProvider
T
getObject(args: Object...): T
getIfAvailable(): T
getIfAvailable(defaultSupplier: Supplier<T>): T «default»
ifAvailable(dependencyConsumer: Consumer<T>): void «default»
getIfUnique(): T
getIfUnique(defaultSupplier: Supplier<T>): T «default»
ifUnique(dependencyConsumer: Consumer<T>): void «default»
iterator(): Iterator<T> «default»
stream(): Stream<T> «default»
orderedStream(): Stream<T> «default»
«interface»
ObjectFactory
T
getObject(): T
java.lang
«interface»
Iterable
T

Figure 2.12 "ObjectProvider" Interface

The next method in the type `ObjectProvider` is `getObject(Object...)` with a variable argument (vararg). The method internally accesses the `getBean(…)` method of the `BeanFactory`—that is, the base type of `ApplicationContext`. This method will return an instance of a bean if it exists or, in the case of a prototype bean, will create a new bean, introducing the named arguments into either the constructor or factory method. If there is no bean of the type, `getBean(…)` will throw an exception.

We've just seen the `getIfAvailable()` method. It returns the bean if it exists and returns `null` if it doesn't exist. That is, `null` is used as an indication that this bean didn't exist. If there was more than one bean matching this type, then again there is a `NoUniqueBeanDefinitionException`.

`getIfAvailable(…)` is an overloaded method where a `Supplier` is allowed. This is a default method, so the interface implements it. `getIfAvailable(Supplier)` first calls the basic method `getIfAvailable()`, and we just discussed that there are three possible results: we get a result back, get no result back—in which case, it's `null`—or get an exception if there's more than one bean. The moment the requested bean is available, that specific bean is returned; otherwise, if the result of `getIfAvailable()` was `null`, the `get()` method on the `Supplier` is called, and that's the result. This is similar to the `orElseGet(Supplier)` method of `Optional`.

In addition, `ifAvailable(Consumer)` is a default method related to the `Optional` method `ifPresent(Consumer)`. That is, if there is a bean and exactly one bean, then the `Consumer` is called on that bean; no result is returned.

In Section 2.7.3, we discussed in abstract that you can specify a base type and ask for all subtypes and implementations. That is, there can potentially be multiple candidates. The `getIfUnique(…)` methods return the candidate if there is only one candidate. If there is no candidate or no multiple candidates, `getIfUnique(…)` returns `null`. While the previous `ifAvailable(…)` methods are also found at `Optional`, these `getIfUnique(…)` methods are specifically for when you have multiple candidates.

There is also a default method, `getIfUnique(Supplier)`, which calls `getIfUnique()`. If the result is `non-null`, the exact result is returned; if the result is `null`, the `Supplier` will provide the bean.

Comparable to method `ifAvailable(Consumer)`, the `Consumer` of `ifUnique(Consumer)` is only invoked if there is a unique bean. This method doesn't return a result.

While the methods mentioned revolve around a single bean, `ObjectProvider` can supply multiple beans. Because `ObjectProvider` is an `Iterable`, an `Iterator` can be retrieved. In addition, we have the default methods `stream()` and `orderedStream()`. Exceptions can be thrown in principle, but for the methods `iterator()`, `stream()`, and `orderedStream()`, no exceptions are documented at the signature of the methods. All other methods from `ObjectFactory` and `ObjectProvider` can throw a `BeansException`.

2.7.5 @Order and @AutoConfigurationOrder *

If an `ObjectProvider` supplies all the beans of a type or if a wiring inserts all the beans of a type into a data structure, the order is unspecified. However, the order of these components can be specified, and there are other places where order matters.

Spring declares an annotation `@Order` (and `@AutoConfigurationOrder` for auto-configuration) and an interface `Ordered` to describe an order. This order is defined by a priority. A priority is defined by the value of an integer.

@Order

The priority (the integer value) is set with the annotation `@Order`.[18] Here, we see three examples of thumbnail implementations with different priorities:

```
@Component @Order( 1955 )
class AwtNearestNeighborThumbnail implements Thumbnail …
@Component @Order( 0 )
class DefaultThumbnail implements Thumbnail …
@Component @Order( -1587 )
class AwtBicubicThumbnail implements Thumbnail …
```

---

[18] *https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/core/annotation/Order.html*

The following logic applies to priority:

- The lower the value, the higher the priority.
- The default is `Ordered.LOWEST_PRECEDENCE`, which is $2^{31} - 1$, so `Integer.MAX_VALUE`.

Intuitively, you would probably expect it to be the other way around.

Suppose we have three beans with different priorities, and we want to inject them into a list:

```
@Autowired
List<Thumbnail> thumbnails;
```

Then the component `awtBicubicThumbnail`—because it has the lowest priority—would be first in the list. This would be followed by the default thumbnail, and the "worst" variant, so to speak, with the lowest priority would be `AwtNearestNeighborThumbnail`.

Ordered Interface

Annotations are nice and compact, but they have a disadvantage in practice: the specified values are static. That's why the Spring Framework has an alternative way of specifying values in many places besides annotations—usually via a special interface. For annotation `@Order`, there is a corresponding functional interface `Ordered` with a `getOrder()` method, and this can also supply a priority. The advantage here is that the priority can be determined at runtime based on certain conditions.

Again, here's a fictional example:

```
@Service
public class AwtBicubicThumbnail implements Thumbnail, Ordered {

  @Override
  public byte[] thumbnail( byte[] imageBytes ) { … }

  @Override
  public int getOrder() {
    return faster computer ? low number : higher number;
  }
}
```

`AwtBicubicThumbnail` implements the `Thumbnail` interface as known and additionally the `Ordered` interface. The idea behind `getOrder()` is the following: `AwtBicubicThumbnail` is sophisticated and consumes more computation time. That is, we only want to prefer this implementation and "move it up in priority," so to speak, if our computer is powerful. If we have a fast computer, `AwtBicubicThumbnail` gets a higher priority, so the `Thumbnail` implementation moves to the front of the list at `@Autowired`, for example. A higher priority means a lower number. Of course, we have to see that when there are multiple `Thumbnail` implementations, these numbers don't collide but instead actually form an order—but that's another story.

2.7.6 Behavior in Selected Inheritance Relationships *

In this section, we'll look at a few scenarios of what happens when a Spring class inherits from another class.

@Bean Method in Superclasses/Interfaces

If an `@Configuration` class inherits `@Bean` methods from a superclass, these factory methods are also called. Here's an example:

```
abstract class Thing {
  @Bean UUID uuid() {
    return UUID.randomUUID();
  }
}
@Configuration class Profile extends Thing { }
```

In terms of the factory method, this is equivalent to the following:

```
abstract class Thing { }
@Configuration class Profile {
  @Bean UUID uuid() {
    return UUID.randomUUID();
  }
}
```

In other words, Spring doesn't care if these `@Bean` methods are part of Spring's own managed bean class or if these methods are inherited—as inherited methods, they go into the subtype just as well. But the superclasses have no annotation like `@Configuration` or `@Component`.

The same works for interfaces, by the way, because they can contain default methods:

```
interface Thing {
  @Bean default UUID uuid() {
    return UUID.randomUUID();
  }
}
@Component class Profile implements Thing { }
```

Default methods can also be annotated with `@Bean` in Spring. If the `Profile` class implements the `Thing` interface, the class inherits the default method with the `@Bean` method, so Spring will generate a Spring-managed bean of type `UUID` in that case as well.

@Component Isn't @Inherited

Continuing our discussion on inheritance, it's worth noting that annotations can be inherited to subclasses. If, for example, a superclass is annotated, it's possible in principle that this annotation is also present in the subclass. However, this isn't automatically the case, but `@java.lang.annotation.Inherited`[19] must be set in the base type.

As an example of an inherited annotation, consider `@EnableAutoConfiguration` and the `@SpringBootApplication`, both of which have the meta-annotation `@Inherited`:

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@AutoConfigurationPackage
@Import(AutoConfigurationImportSelector.class)
public @interface EnableAutoConfiguration
```

That is, if a base type is annotated with `@EnableAutoConfiguration`, the implementation of the interface (or an extension of a class) is also recognized as `@EnableAutoConfiguration`.

The `@Inherited` annotation is used in a few places, but it's not used in many places in the Spring Framework. In particular, `@Component` types aren't `@Inherited`. The consequence is that a superclass that is now annotated with `@Component`, `@Repository`, `@Service`, or `@Controller`, for example, won't pass that annotation on to the subclass. And this is especially important if, for example, abstract superclasses are annotated with `@Component`, and then you think that the subclasses will become Spring-managed beans.

So, here's what doesn't work:

```
@Component
abstract class Thing { }

class Profile extends Thing { }

@Component
class Demander {
  // @Autowired Thing thing;
  @Autowired Profile profile;
}
```

The assigned annotation `@Component` isn't inherited. That is, the `Profile` isn't registered as a Spring-managed bean, and because abstract classes aren't registered as components, neither a `Thing` nor a `Profile` is created. Therefore, the `Demander` class can't have either a `Thing` or a `Profile` injected.

---

[19] *https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/annotation/Inherited.html*

@Autowired Features in the Upper Class

Spring includes from the superclass, no matter if it's abstract or not, for example, the `@Bean` methods. Moreover, a superclass might contain an `@Autowired` method or an instance variable, and Spring would initialize those as well.

Here's a small example:

```
abstract class Thing {
  @Autowired ApplicationContext ctx;
  @Autowired void setArgs( ApplicationArguments args ) { }
}

@Component
class Profile extends Thing { }
```

If Spring builds a bean of type `Profile`, the `@Autowiring` is also performed from the superclass.

Note

Of course, it could be that the `Profile` subclass overwrites the `setArgs(…)` setter. The overridden method then doesn't take annotation `@Autowired`. That is, if we override this method and don't set `@Autowired`, there is no wiring. So, we have to annotate `setArgs(…)` with `@Autowired` if the method from the subclass also wants to receive `ApplicationArguments`.

2.8 Beans Lifecycle

We've used Spring-managed beans in many places, and now let's take a closer look at their lifecycle: When are components instantiated, and how can initialization code be run?

Spring creates components via the constructor if, for example, they are annotated with `@Component` and detected via classpath scanning or if `@Bean` methods return instances. If there is a dependency, for example, via constructor injection, what is injected must be constructed first. The order of the initialization is given thereby.

2.8.1 @DependsOn

If there is a dependency on beans, but no wiring makes it explicit, there is another way for the Spring Framework to initialize certain beans before other beans.

With `@DependsOn`,[20] you can initialize one or more beans before another bean:

---

[20] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/context/annotation/DependsOn.html*

```
@DependsOn( { "aBean", "anotherBean" } )
```

The annotation `@DependsOn` is possible for `@Component` annotated types and also for `@Bean` methods. The annotation attribute contains an array of bean names. These beans are initialized first and then our own component follows.

Assuming that a `FileSystem` needs to be initialized prior to a `FileWatcher`, such that the `FileSystem` can be monitored, this dependency can be expressed through the use of `@DependsOn`, given that there is no wiring dependency between them:

- `@DependsOn` as type annotation:

  ```
  @Component
  @DependsOn( “fileSystem” )
  class FileWatcher { … }
  ```

- `@DependsOn` as method annotation:

  ```
  @Bean
  @DependsOn( “fileSystem” )
  FileWatcher fileWatcher() { … }
  ```

Problems with @DependsOn

The use of `@DependsOn` has two problems:

- Only the name is expected from the dependent component. Names are strings, and strings can be misspelled. If the name is wrong, you get a `NoSuchBeanDefinitionException`, and the container doesn’t start up.
- From the outside, `@DependsOn` isn’t necessarily visible, and it’s a similar problem to field injection. In constructor injection, for example, the constructor’s parameter list makes it clear that another component is needed and required. However, `@DependsOn` is declarative. Spring will listen to this annotation, but in a test case, it’s we who must take care to build this other component as well and initialize it beforehand if necessary.

2.8.2 Delayed Initialization (Lazy Initialization)

The container builds the Spring-managed beans all by default, regardless of whether they are needed or not. In addition to these singletons, that is, instances that exist only once, there are also prototype objects, which is another bean scope we’ll talk about more in Section 2.8.10. But what advantage should there be to having the Spring Framework build all components, regardless of whether they are needed or not? The big advantage of this is that if you “survive” the start of the container, then it’s clear that all components could be built without errors, and then there will be no error in the following due to a bean generation. However, this approach leads to a higher start time because not all components are needed at the beginning, but perhaps sometime later or possibly not at all. (Most components are needed in the context of an application, it's only a question of when they are initialized. Components, which aren't necessary, obviously must be removed from the application.)

To reduce the startup time, the point in time at which the components are built can be postponed as far back as possible until the component is needed, which is called *lazy initialization*. In this case, the components are only built, configured, and initialized when they are really needed, and *needed* in this context means that there is a wiring or a retrieval of the component from the context.

Delayed initialization has advantages and disadvantages:

- **Advantages**
  The advantage is the reduced startup time. Memory is reduced if you don't build the objects that aren't needed for the current program to run.
- **Disadvantages**
  If the container builds the objects later, an error can still occur afterward. It's not optimal if the application has been running for some time, then suddenly an error occurs, and the Spring application shuts down.

[+]

Tip: Best Practice

In software development, short startup times play a more significant role during the test and development phase compared to the production system. This is because the application is frequently started and stopped during this phase. As a result, short startup times are critical for efficient testing and development. In such cases, lazy initialization can be used to optimize the startup time. However, it's not recommended to use lazy initialization for a released product.

Delayed Initialization with @Lazy

Spring declares an annotation with `@Lazy` to control late initialization. One use is at the class type:

```
@Lazy @Component
class FileSystem { … }
```

Another use is for factory methods:

```
@Lazy @Bean
UUID appUuid() { … }
```

If the `FileSystem` is `@Lazy`, and there is a wiring to the `FileSystem`, then of course it's needed and built. But another component could be a controller, which reacts only on a web request and uses the `FileSystem`. As long as there are no requests, the `FileSystem` isn't necessary. It's exactly the same with the `@Bean` method: if nobody is currently interested in a UUID, this factory method won't be called.

The annotation `@Lazy` can appear at `@Component` classes and of course also at specializations such as `@Configuration` classes. If `@Configuration` classes contain `@Bean` methods, the following rule applies: IF an `@Configuration` class is `@Lazy`, then this is automatically transferred to each `@Bean` method. `@Lazy` can be turned off for individual methods, which we'll get to in a moment.

If you can't modify the components because, for example, the source code isn't available, another option is to use `@Lazy` at the injection points—that is, at the places annotated with `@Autowired` or `@Inject`. Even then, the component to be injected is initialized with a delay. To find out if there was an initial access to the injection point, Spring uses a proxy internally.

Global Delayed Initialization

Annotation `@Lazy` is always local, which means it can be decided individually for each component whether it should be initialized with a delay or not. Configuration property `spring.main.lazy-initialization` offers an alternative. If it's set to `true`, the lazy mode automatically applies to all components.

Another way is explicitly offered by Spring Boot when building the container via methods:

- `SpringApplication`: `setLazyInitialization(true)`
- `SpringApplicationBuilders`: `lazyInitialization(true)`

If everything is configured *lazy*, for example by this switch, then the delayed initialization for certain Spring-managed beans can be disabled with `@Lazy(false)`. This would be important, for example, if `@Lazy` is on a configuration (and we said that `@Lazy` is then applied to all `@Bean` methods). So, it's possible to define that an instance should be built directly for individual `@Bean` methods.

2.8.3 Bean Initialization Traditional

When a component is initialized, numerous actions take place. For instance, a Spring-managed bean is a class that is automatically instantiated by the container through reflection. We create objects using factory methods. After that, the Java virtual machine (JVM) reserves memory for instance variables when a new object is created. The class and instance variables are "zeroed," and the constructor initializes the instance variables. The superclass constructor is called by `super(…)`. This is the default initialization from the JVM and typical for any instance. This lifecycle is significantly refined by the Spring Framework.

Restricted Initialization in the Constructor

Let's assume the following implementation in the constructor:

```
@ShellComponent
public class FsCommands {

  private final Logger log = LoggerFactory.getLogger( getClass() );

  @Autowired private FileSystem fs;

  public FsCommands() {
    if ( fs.getFreeDiskSpace() < 1_000_000 )
      log.info( "Low disk space!" );
  }

  // …
}
```

The example uses field injection to put a `FileSystem` instance into an `FsCommands` instance. The constructor wants to check if there is enough free memory.

However, this implementation is *not* correct and leads to a `NullPointerException`. The reason is in the lifecycle of the component. First, the Spring Framework calls the constructor of `FsCommands`, but the field injection takes place much later. That is, the instance variable `fs` isn't initialized; Spring can't build the object and allocate the variable before the constructor call.

There are two ways to solve the problem:

- The first possibility is a constructor injection, which is the better variant anyway. If you want a `FileSystem` directly with constructor injection, you can access the `FileSystem` directly in the constructor.
- Special callback methods can be defined, which are called after wiring.

Lifecycle Callbacks

Spring provides a comprehensive mechanism for building and initializing components, which enables the execution of specific operations during the creation or removal of a bean from the container. This is made possible through Spring's implementation of *lifecycle callbacks* on components, which are special methods that can be provided or marked up as Spring-managed beans and called as part of the lifecycle. The most common approach to this is the declarative approach using annotations. While there is also an older approach of implementing certain interfaces, using Spring's special interfaces is generally not recommended, although it's still briefly mentioned for the sake of completeness.

Jakarta Annotations API

The *Jakarta Annotations API* declares the two annotation types `@PostConstruct` and `@PreDestroy`. This annotations API was integrated in Java 6, but removed again in Java 11. For this reason, Spring automatically references the current Jakarta Annotations API via a starter using the Maven dependency `jakarta.annotation:jakarta.annotation-api`.

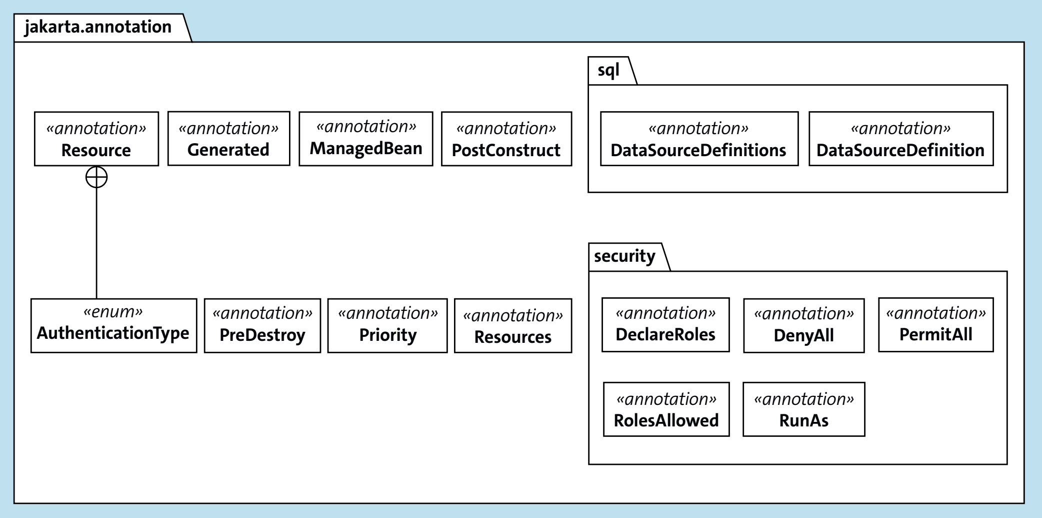

jakarta.annotation
«annotation» Resource
«annotation» Generated
«annotation» ManagedBean
«annotation» PostConstruct
sql
«annotation» DataSourceDefinitions
«annotation» DataSourceDefinition
«enum» AuthenticationType
«annotation» PreDestroy
«annotation» Priority
«annotation» Resources
security
«annotation» DeclareRoles
«annotation» DenyAll
«annotation» PermitAll
«annotation» RolesAllowed
«annotation» RunAs

Figure 2.13 Contents of the “jakarta.annotation” Package

The Jakarta Annotations API (*https://jakarta.ee/specifications/annotations/2.0/api-docs/jakarta.annotation/module-summary.html*) consists of a package `jakarta.annotation` (see Figure 2.13) and two subpackages. The annotations are used in other contexts. In our case, we’re only interested in the annotations `@PostConstruct` and `@PreDestroy`. There is also the annotation `@Priority`, which can be used to set priority and is an alternative to the `@Order` annotation described in Section 2.7.5.

@PostConstruct and @PreDestroy

`@PostConstruct` and `@PreDestroy` are annotations that you can use for methods. That is, Spring will recognize the annotations and then automatically call these annotated methods in the lifecycle:

- **`@PostConstruct`**
  Used for a method called after dependency injection to perform initialization.
- **`@PreDestroy`**
  Used for methods that generally release resources before the Spring-managed bean is removed from the container. Normally, cleanup comes into such methods when, for example, you want to release resources associated with this Spring-managed bean.

Several methods can carry the lifecycle annotations, in which case, there is no information about the order.

The methods have a few things in common:

- They have no parameters (except for Jakarta Enterprise Beans interceptors).
- They aren't `static`.
- They return nothing; the return type is `void`.
- They don't raise any checked exceptions. (Of course, they could in principle throw `RuntimeExceptions`.)
- They can have any visibility. (These methods are called automatically via reflection.)

An Example with @PostConstruct

Consider an example that uses the `@PostConstruct` annotation to address the issue of checking the number of free bytes during component construction, as an alternative to constructor injection:

```
@ShellComponent
public class FsCommands {

  private final Logger log = LoggerFactory.getLogger( getClass() );

  @Autowired private FileSystem fs;

  @PostConstruct
  public void checkDiskSpace() {
    if ( fs.getFreeDiskSpace() < 1_000_000 )
      log.info( "Low disk space!" );
  }
  // …
}
```

Spring calls the default constructor, and sometime later, any `@Autowired` will take place, and the `FileSystem` will be injected. We put the query for the number of free bytes in a separate method `checkDiskSpace()`. There we can fall back on the `FileSystem` because `@PostConstruct` is called at the end after the wiring.

Note: The @PostConstruct Problem

In contrast to constructor injection, the invocation of a method annotated with `@PostConstruct` during initialization isn't explicitly apparent in the source code. This makes it possible to misuse the API, such as failing to call the `@PostConstruct` method manually during testing. When a constructor injection is available, it's advisable to use it instead.

`@PostConstruct` can be used well for later validation. For example, an `@PostConstruct` method could be used well for the following scenario: with setter-injection, several optional objects are injected, but some initializations exclude each other.

At this point, we briefly look ahead to Chapter 3, Section 3.3.1. These `*Runners` are much better suited to run code after the object graph has been completely constructed. `@PostConstruct` isn't well suited for this because it only runs locally when its own component is ready; later, there may be more initializations, which means the `@PostConstruct` method can't access everything yet. If you want to run programs after the container got started, for example, command-line programs, then this code is much better off in a `*Runner`.

@PreDestroy

The methods annotated with `@PreDestroy` are executed whenever the container wants to remove this component. Technically, this works in Java with a *shutdown hook*, which executes a `Runnable` when the JVM shuts down. These "hooks" work excellently in practice, even if `System.exit(…)` terminates the application.

Spring independently registers such a shutdown hook, and `@PreDestroy` clearing operations then run in it. Here's a small example: the service `FileSystem` should make a log output with the free gigabytes in the file system at the end when the application stops:

```
@Service
class FileSystem {
  …

  @PreDestroy
  void stop() {
    log.info( "Exiting application with {} free GiB",
              DataSize.ofBytes( getFreeDiskSpace() ).toGigabytes() );
    // log.info( Thread.currentThread().toString() );
    // Thread[SpringApplicationShutdownHook,5,main]
  }
}
```

If you look at the executing thread, you can read the `SpringApplicationShutdownHook`, which is the "hook" that Spring uses to execute the `@PreDestroy` callback methods at the end.

@Bean Annotation Attribute [init|destroy]Method

While we can work directly with annotations `@PostContruct` and `@PreDestroy` for regular `@Component` classes, this becomes a bit more difficult with methods produced by beans because the bean methods are subject to our control, and Spring Framework has nothing to do with that. Lifecycle methods can basically be specified for `@Bean` as well; the annotation type shows this:[21]

```
@Target({ElementType.METHOD, ElementType.ANNOTATION_TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface Bean {

  @AliasFor("name") String[] value() default {};
  @AliasFor("value") String[] name() default {};
  boolean autowireCandidate() default true;
  String initMethod() default "";
  String destroyMethod() default AbstractBeanDefinition.INFER_METHOD;

}
```

`@Bean` has a name, which we can specify via `value` or `name`, as explained in Section 2.7.1. In addition, you can control whether beans are an `autowireCandidate`, which they are by default, of course.

Then there are the two annotation attributes `initMethod` and `destroyMethod` to specify the lifecycle method: `initMethod` specifies the name of a method to be called after the bean is initialized. Here is the “raw” name of the method, that is, without round brackets because we can’t specify overloaded methods anyway. (For `@PostConstruct` and `@PreDestroy`, the methods generally have no parameters either.) Equivalently, with `destroyMethod`, we specify a method to be called before the container removes that component.

Note

`destroyMethod` has an interesting preassignment with `AbstractBeanDefinition.INFER_METHOD`. The documentation calls this *destroy method inference*: If our bean class that we use to instantiate and return has a public `close()` or `shutdown()` method, then the Spring container will automatically call those methods. Of all the wild things Spring does, this is worth shaking your head at. So we need to keep in mind if our bean has a public parameterless `close()`/`shutdown()` method and whether or not it should be called. The logic is a little more complicated:

1. First, a `close()` method is searched for. If it exists, it’s called.
2. If Spring Framework doesn’t find a `close()` method but finds a `shutdown()` method, it will call the `shutdown()` method.
3. If both `close()` and `shutdown()` are found, then only `close()` is called and not both.

---

[21] In Spring Framework 6, the `@Deprecated` attribute `Autowire autowire() default Autowire.NO` has been removed.

2.8.4 InitializingBean and DisposableBean *

In modern programs, we always annotate methods with `@PostConstruct` and `@PreDestroy` to be called automatically as part of the lifecycle. Because annotations were only introduced in Java 5, only interfaces were allowed to "mark" callback methods at that time.

Since Spring 1.0, there have been two special interfaces in the `org.springframework.beans.factory` package:

```
public interface InitializingBean {
  void afterPropertiesSet() throws Exception;
}
public interface DisposableBean {
  void destroy() throws Exception;
}
```

These two interfaces can be implemented by our components, and the callback methods are then called in the lifecycle.

Note

A lifecycle for a component can be defined simultaneously using annotations and callback interfaces. Then, `@PostConstruct` annotated methods are called before `afterPropertiesSet()`, and `@PreDestroy` methods are called before `destroy()`.

These two interfaces are a relic of the past and should not be used in modern Spring applications. These interfaces are invasive, which means that our components get a technical dependency that is unnecessary. That is why this variant is no longer recommended today.

2.8.5 Inheritance of the Lifecycle Methods

The lifecycle methods we just declared—annotated or overridden—can be inherited and will then be called as inherited methods during initialization or shutdown as well.

Here's a small, contrived example:

```
class IOResource {
  @PostConstruct
  void init() {
    …
  }
}

interface Closeable {
  @PreDestroy
  default void close() {
    …
  }
}

@Service
class FileSystem extends IOResource implements Closeable {
}
```

If Spring initializes the `FileSystem`, `init()` and `close()` are called automatically in the lifecycle.

2.8.6 *Aware Interfaces *

Interfaces `InitializingBean` and `DisposableBean` are only two selected interfaces. The Spring Framework declares quite a few additional interfaces that are called as part of the initialization process. Annotations for “marking up” have been around since Java 5; before that, you couldn’t just “mark up” methods except by prescribing them through interfaces. (The Spring makers could have chosen magic method names, but that’s even more obscure than declaring that method in an interface.)

That’s why the Spring Framework declares a set of interfaces that our components can implement; then the Spring Framework calls these methods and passes certain objects. We can remember these objects and do something with them later. Here are a few examples:

- `ApplicationContextAware`: `setApplicationContext(ApplicationContext applicationContext)`
- `BeanNameAware`: `setBeanName(String name)`
- `EnvironmentAwaresetEnvironment(Environment environment)`
- `BeanClassLoaderAwaresetBeanClassLoader(ClassLoader classLoader)`

We implement `ApplicationContextAware` if we want to have access to the `ApplicationContext`. Then our component will have a `setApplicationContext(…)` method, and Spring Framework will pass us the `ApplicationContext` in the method. We can remember that for later and call methods on the `ApplicationContext` in the future. From today’s point of view, this isn’t necessary because we can have an `ApplicationContext` injected with `@Autowired`.

Here’s another example: we have interface `BeanNameAware`, which gives us the name of the bean. Then, when we implement the interface, we only need to override the `setBean(…)` method.

An `Environment` provides the possibility to access configuration properties, which are stored, for example, in the *application.properties* file. If we want access to the `Environment`, we could implement a `setEnvironment(…)` method. The Spring Framework will then provide us with an instance on the `Environment`. In modern Spring applications, there are alternatives, such as an `@Autowired Environment`. We'll look at this data type in more detail in Chapter 3, Section 3.2.

Another example is the `setBeanClassloader(…)` method from `BeanClassLoaderAware`. If a class implements this interface, then Spring passes us the current class loader during the initialization phase.

A common feature of the interfaces is that they end in `-Aware`. There is one marker interface (an interface without implementation) `Aware`[22] and more than 300 "implementations"—that's quite a lot. Through various callback methods, we can make ourselves "aware" of all sorts of things.

BeanFactoryAware versus Injection

If a class implements the `BeanFactoryAware` interface, it can accept a `BeanFactory` via a setter. This is no longer necessary in modern Spring applications because a program can have a `BeanFactory` (and, of course, all existing subtypes, such as `DefaultListableBeanFactory`) injected with an injection.

DefaultListableBeanFactory

`DefaultListableBeanFactory`[23] is a complex type with a large number of type relationships, which, fortunately, we don't need to look at in detail. Following are two capabilities of `DefaultListableBeanFactory` we want to take a closer look at:

- Beans can be registered manually.
- Subsequent wiring can be carried out.

For registering new Spring-managed beans, we had never programmed anything before. The variants about annotated classes, annotated factory methods, and classpath scanning are all declarative, along with the extension with `@Import`. The option to build and register beans by hand can be useful. To demonstrate this, let's consider a brief example of how to manually register beans using the `registerBeanDefinition(…)` method of the `DefaultListableBeanFactory`:

```
class Location {
  double longitude, latitude;
  public void setLongitude( double longitude ) {
```

---

[22] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/beans/factory/Aware.html*

[23] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/beans/factory/support/DefaultListableBeanFactory.html*

```
    this.longitude = longitude;
  }
  public void setLatitude( double latitude ) {
    this.latitude = latitude;
  }
}
@Component class LocationRegistration {
  LocationRegistration( DefaultListableBeanFactory beanFactory ) {
    BeanDefinition definition = BeanDefinitionBuilder
        .genericBeanDefinition( Location.class )
        .addPropertyValue( "latitude", 51.5364099 )
        .addPropertyValue( "longitude", 7.41571 )
        .getBeanDefinition();
    beanFactory.registerBeanDefinition( "location", definition );
  }
}

@Component @DependsOn( "locationRegistration" )
class LocationUser {
  public LocationUser( Location location ) { … }
}
```

The `Location` class has no reference to the Spring Framework. But the goal is to build a Spring-managed bean from it. This class deliberately has setters for the configurations because they will be called by the Spring Framework in the following.

The second class is a Spring-managed bean that requests the `DefaultListableBeanFactory` via constructor injection. The Spring Framework stores bean definitions of type `BeanDefinition`,[24]and we can build these ourselves and thus store a Spring-managed bean in the framework. If we use class `BeanDefinitionBuilder`, then the new bean can be configured with the Fluent API:

- `genericBeanDefinition(…)`
  The method gets a type token as information regarding which instance should be built; in our case, a `Location` object should be created.
- `addPropertyValue(…)`
  This sets the two properties `latitude` and `longitude`. That's why the two setters were declared in `Location`.

The call `getBeanDefinition()` completes the builder, and the result is a `BeanDefinition` object. This is passed to `registerBeanDefinition()` along with a bean name; we want our bean to be called "location".

---

[24] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/beans/factory/config/BeanDefinition.html*

So, in the end, there is an additional component in the context, which can be injected into the code. You can see this with `LocationUser`, which requests the `location` via constructor injection.

The question is whether the usage is particularly well implemented. The problem is that this component can be built at some point in the application lifecycle. The lifecycle isn't clear because the component doesn't appear until `LocationRegistration` has called the `registerBeanDefinition(…)` method. When Spring builds a `LocationUser` and looks for a `Location` in context, `Location` may or may not have been built *before* `LocationUser` was built—and that's a problem because then there is an exception.

The annotation `@DependsOn` can specify that `LocationUser` is dependent on a bean named `locationRegistration`. This expresses a certain order, but overall, this is a pretty shaky construct, and you won't want to use it that way.

2.8.7 BeanPostProcessor *

The `*Aware` callback methods have a special feature, just like our callback methods: they are our *own* methods implemented on *our* data types. For example, if a method is annotated with `@PostConstruct`, then the program has made the decision that additional initialization is needed. The same is true with Spring interfaces: if a class implements them, the Spring Framework calls the methods.

Away from our methods, there are things that Spring Framework should do without us having to provide callback methods for them ourselves. A few examples are given here:

- Wiring is one of those things: when values are injected, we declaratively write an `@Autowire`, and that isn't a callback method. In addition, resources can be injected.
- Spring often builds a proxy around the component that handles validations, for example.
- Our own methods can be called regularly via a timer.

The peculiarity in these cases is that no callback methods are needed, but the Spring Framework does something with the object during the initialization phase.

For such tasks, Spring provides interface `BeanPostProcessor`.[25] It's one of the most important interfaces in the entire Spring Framework. Almost everything magical that happens (and that includes wiring) is programmed with a `BeanPostProcessor`. Every component that Spring builds as a bean runs through every registered `BeanPostProcessor`, and it can look at the object and decide the following:

- At one point or another, a value must be set.
- Some method must be called.
- The object is wrapped with a proxy.

---

[25] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/beans/factory/config/BeanPostProcessor.html*

These types of tasks are performed by a `BeanPostProcessor`.

Methods of the BeanPostProcessor Interface

The interface is relatively simple, consisting of only two default methods. The callback methods are called, either before or after the initialization routine on the beans:

```
public interface BeanPostProcessor {

  @Nullable
  default Object postProcessBeforeInitialization(   ❶
      Object bean, String beanName
  ) throws BeansException {
    return bean;                                     ❸
  }

  @Nullable
  default Object postProcessAfterInitialization(    ❷
      Object bean, String beanName
  ) throws BeansException {
    return bean;
  }

}
```

❶ Property values were set, but lifecycle initializations (e.g., `@PostConstruct`, `afterPropertiesSet`, or assigned `init-method`) weren't running yet.

❷ As ❶, only lifecycle initializations are completed.

❸ Both methods can return a proxy (wrapper).

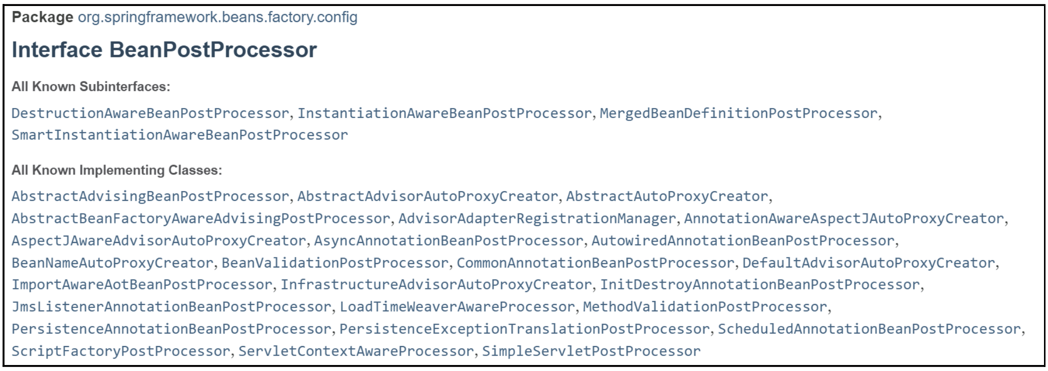
So while the `postProcessBeforeInitialization(…)` method runs before the lifecycle initializations, `postProcessAfterInitialization(…)` is called after the initialization. A `BeanPostProcessor` can build a proxy around this input bean; the default implementation simply returns the bean input.

This implementation of `BeanPostProcessor` is completely separate from components. This means that a `BeanPostProcessor` can work on different Spring-managed beans.

A Sample Implementation of BeanPostProcessor

The Spring Framework uses a large number of `BeanPostProcessor` implementations. If you look at the Java documentation, you can see a few of these classes (see Figure 2.14).

There are significantly more classes, but these are the ones that are publicly shown via the Java documentation.



Package org.springframework.beans.factory.config

Interface BeanPostProcessor

All Known Subinterfaces:

`DestructionAwareBeanPostProcessor`, `InstantiationAwareBeanPostProcessor`, `MergedBeanDefinitionPostProcessor`, `SmartInstantiationAwareBeanPostProcessor`

All Known Implementing Classes:

`AbstractAdvisingBeanPostProcessor`, `AbstractAdvisorAutoProxyCreator`, `AbstractAutoProxyCreator`, `AbstractBeanFactoryAwareAdvisingPostProcessor`, `AdvisorAdapterRegistrationManager`, `AnnotationAwareAspectJAutoProxyCreator`, `AspectJAwareAdvisorAutoProxyCreator`, `AsyncAnnotationBeanPostProcessor`, `AutowiredAnnotationBeanPostProcessor`, `BeanNameAutoProxyCreator`, `BeanValidationPostProcessor`, `CommonAnnotationBeanPostProcessor`, `DefaultAdvisorAutoProxyCreator`, `ImportAwareAotBeanPostProcessor`, `InfrastructureAdvisorAutoProxyCreator`, `InitDestroyAnnotationBeanPostProcessor`, `JmsListenerAnnotationBeanPostProcessor`, `LoadTimeWeaverAwareProcessor`, `MethodValidationPostProcessor`, `PersistenceAnnotationBeanPostProcessor`, `PersistenceExceptionTranslationPostProcessor`, `ScheduledAnnotationBeanPostProcessor`, `ScriptFactoryPostProcessor`, `ServletContextAwareProcessor`, `SimpleServletPostProcessor`

Figure 2.14 Some “BeanPostProcessor” Implementations from the Javadoc

An essential `BeanPostProcessor` takes care of the wiring; the class is called `AutowiredAnnotationBeanPostProcessor` and can be seen in the screenshot. The implementation, after initialization is done, will look in the bean for `@Autowired` properties and, if they are present, initialize them with reflection.

ApplicationContextAwareProcessor

We can examine the source code of `ApplicationContextAwareProcessor` to gain a clear understanding of its implementation.[26]

```
class ApplicationContextAwareProcessor implements BeanPostProcessor {

    private final ConfigurableApplicationContext applicationContext;

    private final StringValueResolver embeddedValueResolver;

    public ApplicationContextAwareProcessor(
               ConfigurableApplicationContext applicationContext) {
        this.applicationContext = applicationContext;
        this.embeddedValueResolver =
          new EmbeddedValueResolver(applicationContext.getBeanFactory());
    }


    @Override
    @Nullable
    public Object postProcessBeforeInitialization(Object bean,
                                                  String beanName) throws BeansException {
        if (!(bean instanceof EnvironmentAware
```

---

[26] *https://github.com/spring-projects/spring-framework/blob/main/spring-context/src/main/java/org/springframework/context/support/ApplicationContextAwareProcessor.java*

```
              || bean instanceof EmbeddedValueResolverAware
              …
              || bean instanceof ApplicationContextAware
              || bean instanceof ApplicationStartupAware)) {
              return bean;
          }

          invokeAwareInterfaces(bean);
          return bean;
      }

      private void invokeAwareInterfaces(Object bean) {
          if (bean instanceof EnvironmentAware) {
              ((EnvironmentAware) bean)
                 .setEnvironment(this.applicationContext.getEnvironment());
          }
          if (bean instanceof EmbeddedValueResolverAware) {
              ((EmbeddedValueResolverAware) bean)
                 .setEmbeddedValueResolver(this.embeddedValueResolver);
          }
          …
          if (bean instanceof ApplicationContextAware) {
              ((ApplicationContextAware) bean)
                 .setApplicationContext(this.applicationContext);
          }
      }
}
```

Listing 2.20 Implementation of the "ApplicationContextAwareProcessor" Class

The class implements `BeanPostProcessor` and nothing else. The parameterized constructor is called by the Spring Framework itself and provides the implementation with information that is needed later.

`postProcessBeforeInitialization(Object bean, String beanName)` is overridden, and the implementation checks if the `bean` has a certain type. If the bean doesn't have one of the selected types, it's returned directly. However, if an interface like `EnvironmentAware`, `EmbeddedValueResolverAware` and so on has been implemented, then method `invokeAwareInterfaces(…)` is called.

`invokeAwareInterfaces(…)` checks again: if the instance is of type `EnvironmentAware`, then it's upgraded via type conversion, and the `setEnvironment(…)` method is called on the bean. This is the function of the `*Aware` interfaces: We get something. Spring just has to pass us the `Environment`. The `BeanPostProcessor` gets that itself from the context. The other types work similarly.

Large Number of Calls and Sequences

There are many implementations of the `BeanPostProcessor` interface, and you must consider that each bean runs through each `BeanPostProcessor`. That is, we have relatively many calls of these callback methods, namely, n beans and m `BeanPostProcessor`, then $2 \times n \times m$ calls.

Because there are many `BeanPostProcessor` implementations, the order may matter. It may be that something should be initiated first and then a proxy should be placed around the object later—and not vice versa. This order can be defined in two ways:

- If the `BeanPostProcessor` class implements the `Ordered` interface (the type was explained in Section 2.7.5), the order can be specified.
- A `BeanPostProcessor` can be added to an `ApplicationContext` manually. The order of these additions results in the order in which the `BeanPostProcessor` methods are called.

2.8.8 Register Spring-Managed Beans Somewhere Else

Spring is a very important enterprise framework that manages the components, injects them, connects to infrastructure technology, and so on. Of course, there are other frameworks, and they might be interested in our Spring components. In the past, we've taken Java objects that might have come from another framework and sort of brought them into the Spring universe.

Now, let's consider the scenario where we want to use Spring components in other parts of the application. This is a straightforward process, thanks to the `BeanFactory`. The `BeanFactory`, which is our `ApplicationContext`, offers several methods for retrieving and querying beans. Therefore, obtaining Spring beans is a relatively simple task.

Another useful method of the `BeanFactory`[27] is `getBeansWithAnnotation(…)`, which allows us to retrieve Spring-managed beans that have a specific annotation. This is particularly helpful when we need to transfer special annotated types to other frameworks. Here's a brief example:

```
@Controller
@Retention( RUNTIME )
@interface HttpController {
  String value() default "/";
}

@HttpController( "/date" )
class DateHandler implements HttpHandler {
  public void handle( HttpExchange t ) throws IOException {
```

---

[27] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/beans/factory/BeanFactory.html*

```
    t.sendResponseHeaders( HttpURLConnection.HTTP_OK, 0 );
    try ( OutputStream os = t.getResponseBody() ) {
      os.write( LocalDate.now().toString().getBytes() );
    }
  }
}

@SpringBootApplication
public class Serve {
  public static void main( String[] args ) throws IOException {
    var server = HttpServer.create( new InetSocketAddress( 8000 ), 0 );
    SpringApplication.run( Serve.class, args )
        .getBeansWithAnnotation( HttpController.class )
        .forEach( ( __, handler ) -> server.createContext(
            handler.getClass().getAnnotation( HttpController.class ).value(),
            (HttpHandler) handler ) );
    server.start();
  }
}
```

The example implements a web server with endpoints. A Java SE class is used because the Java Development Kit (JDK) includes a web server. There are three types:

- `HttpController`
  The `HttpController` annotation type is meta-annotated with `@Controller`. The annotation should be visible at runtime, and it has a `value` for the path the controller should listen on.
- `DateHandler`
  The `DateHandler` class is the only controller and is annotated with `HttpController`; the path specification expresses that the `DateHandler` takes care of the `/date` path. In addition, the class implements the `HttpHandler`[28] interface from the JDK; when there is an HTTP request, the `handle(HttpExchange)` method is later called as a callback. The status code is set to `OK`, and `0` is set for the number of bytes—as an expression that the exact length is still unknown. The program writes the date in the `OutputStream`.
- `Serve`
  The `Serve` class contains the interesting part in the `main(…)` method. First, the server is built, and the port is passed. Later, the server is launched with `start(…)`. The Spring container is started, and `run(…)` returns a `BeanFactory`. With this, `getBeansWithAnnotation(…)` queries all Spring-managed beans that are annotated with `@HttpController`. The result is a `Map` that our program iterates over. The key is the name of the component, and the associated value is the Spring-managed bean.

---

[28] *https://docs.oracle.com/en/java/javase/17/docs/api/jdk.httpserver/com/sun/net/httpserver/HttpHandler.html*

> From the Spring-managed bean, we extract the annotation attribute and use the path and the bean to call the `createContext(…)` method from `HttpServer`.[29] Finally, the server is started.

In this example, we could give Spring components to another framework, in this case, the HTTP server from JDK. It's rare that you need something like this, but especially if you want to include foreign frameworks in Spring, it might be necessary.

2.8.9 Hierarchical Contexts *

So far, we've always used a context that contains all Spring-managed beans; and it's precisely these beans that can be injected anywhere in the application. In addition to this global context, *hierarchical contexts* can be formed, that is, parent-child relationships. The special feature of heirarchical contexts is the ability to form closed modules, that is, self-contained areas. The areas have their own components, which aren't visible in another context.

Such closed contexts are useful. One example is the *onion architecture*, where we have a clear direction that outside components can see the components inside, but not vice versa. Because if we think about domain-driven design, the domain in the core is most important, and it should be completely technology independent. The layers are arranged around the outside, for example, the infrastructure. The outer layers may use the inner components, but the inner components know nothing about the outer infrastructure.

Something like this can be implemented well with a hierarchical context, so that the innermost components are the base, and the outer components are children that can see components of their parents (in the core), but not vice versa. In a web application, such hierarchical contexts are interesting because different web contexts are isolated, and one web context should not see the components of another web context. Of course, there should be commonalities provided by a parent context, but still, each web application should use its own beans.

Phase 1: Separate Types

The first step is to separate the types. Therefore, the following example class contains two configurations with an `@ComponentScan` for completely different packages:

```
package abc.hierarchy;

public class MainApplication {

  // parent context
```

---

[29] *https://docs.oracle.com/en/java/javase/17/docs/api/jdk.httpserver/com/sun/net/httpserver/HttpServer.html*

```
  @Configuration( proxyBeanMethods = false )
  @ComponentScan( "abc.hierarchy.parent" )
  public static class ParentConfig { }

  // child context
  @Configuration( proxyBeanMethods = false )
  @ComponentScan( "abc.hierarchy.child" )
  public static class ChildConfig { }

}
```

The configurations could have been somewhere else, but the point is to use the `@ComponentScan` annotation. Later, when the `ParentConfig` configuration is specified, everything will be included under `abc.hierarchy.parent`, and for `ChildConfig`, everything will be included under the `abc.hierarchy.child` package. Later, we'll put the parent components under `abc.hierarchy.parent` and will see that types in these packages can be requested by the children, but not vice versa. We'll extend the class a bit in a moment.

As an example, let's look at what `Parent` and `Child` look like:

```
package abc.hierarchy.parent;

@Component
public class Parent {
//  @Autowired Child child;  ☠
}
package abc.hierarchy.child;

@Component
public class Child {
  @Autowired Parent parent;
}
```

The `Parent` class in the `abc.hierarchy.parent` package is detected via classpath scanning in `ParentConfig` and `Child` via `ChildConfig`. Later, when we configure the hierarchical context, `Parent` won't be able to access `Child`, but vice versa.

Next, we need to configure our application, and we usually do that via `@SpringBootApplication`. However, we can't use this because the classpath scanning would now detect all components by default, so we would have nothing at all from the separation with the `parent` and `child`. That's why we could do something different:

```
package abc.hierarchy;

@SpringBootConfiguration
@EnableAutoConfiguration
```

```
public class MainApplication {

  …
  static class *Config { }

  public static void main( String[] args ) { … }
}
```

Instead of `@SpringBootApplication`, `@SpringBootConfiguration` and `@EnableAutoConfiguration` are used, but classpath scanning is missing. Through the nested `*Config` classes we discussed earlier, the packages are scanned. Next, we need to build the context in the `main(…)` method.

Phase 2: Hierarchical Contexts via SpringApplicationBuilder

To build a hierarchical context, we can't use the `SpringApplication` class, but instead we use a class that we described in Section 2.1.3—`SpringApplicationBuilder`—only with that class is it possible to create hierarchical contexts. Three methods of `SpringApplicationBuilder` define the hierarchy:

- `parent(Class<?>... sources)`
  Sets the parent context.
- `child(Class<?>... sources)`
  Sets the children (defaults are taken from parents). Back comes the new context from the child!
- `sibling(Class<?>... sources)`
  Places siblings equally under the parent context.

It's important to pay attention to the order because you usually configure the contexts "fluently." What you get back from the `child(…)` method is a new context from the child. Therefore, if you want to change anything in the parent context, you have to be careful not to change the child by mistake.

If we configure siblings, children can be placed *next to each other* under a common parent. This means that the API provides different modeling options:

- Under one context, there is another context, and under that another context—so something like grandparents, parents, and children.
- Under the parent context, there are various equivalent siblings, all of which take components from the parent context. But the siblings have their own components locally.

To provide an example, let's demonstrate how to have `ChildConfig` as a child configuration of `ParentConfig`:

```
new SpringApplicationBuilder( /* ParentConfig.class */ )
    // .sources( ParentConfig.class )
    .parent( ParentConfig.class )
    .child( ChildConfig.class )
    //  .sibling( AnotherChildConfig.class )
    .run( args );
```

The use of the `parent(…)` method isn't necessary. We could have also used `ParentConfig` in the parameterized constructor of `SpringApplicationBuilder` or specified it at `sources(…)`.

If we wanted to have siblings for the child, then we would pass a different child configuration using the `sibling(…)` method. Again, note that the order is important! Reversing the configuration and writing `sibling(AnotherChild.class).child(ChildConfig.class).parent(ParentConfig.class)` would cause complete chaos.

2.8.10 Singleton and Prototype Stateless or Stateful

The components we've built are singletons. This refers to objects that exist only once in the framework, which is something that Spring does automatically. The moment we have an `@Component` or an `@Bean` method, the framework builds those components only once and puts them in the container.

Besides singletons, there is a second type of component that Spring can build, and that is the prototype, as discussed earlier in this chapter. A prototype is created only when it's needed, and on each new request, there is a new object. A typical example is a shopping cart: with different user sessions within a web server, each user should individually get their own shopping cart.

When working with objects, we must always be careful not to accidentally change object states. When using singletons, it's best to keep the singletons *immutable*. This means that there are states (which are often references to our befriended objects), but these states won't be changed later. If you want to change states, then you should make sure that you only perform thread-safe accesses and changes. Because it could happen at any time that several threads access a singleton. It's best not to use mutating instance variables with Spring components and concurrent accesses.

With a prototype object, you don't have this problem in the first place because different clients always get new objects and can change them according to their wishes. Prototypes are usually *mutable* (i.e., changeable). We'll only have a problem with a prototype object if we have a shared state via static variables, for example. Because static variables are a special case, and not too common, we can ignore that.

2.9 Annotations from JSR 330, Dependency Injection for Java *

At the beginning (with J2EE) there was no dependency injection. A further development, Java EE 5 (mid-2006), used annotations, but a general injection mechanism wasn't yet implemented. Things got interesting in May 2009 when Bob Lee of Google and Rod Johnson of Spring Source opened JSR 330, *Dependency Injection for Java* (*https://jcp.org/en/jsr/detail?id=330*). At that time, there were various IoC containers, and JSR 330 was intended to create a certain set of annotations that all containers could agree on, which were, in that context, annotation types `@Inject`, `@Named`, `@Qualifier`, `@Scope`, and `@Singleton`, as well as a `Provider` interface (*https://jakarta.ee/specifications/dependency-injection/2.0/apidocs/*; see Figure 2.15).

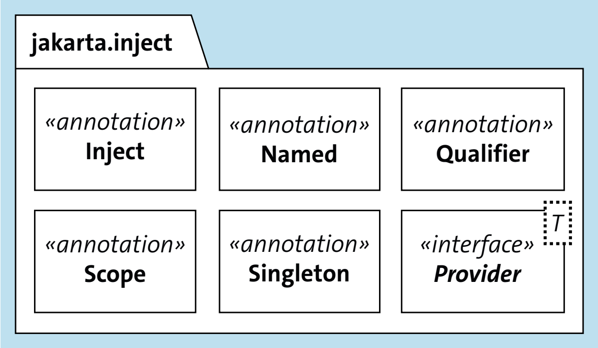


Figure 2.15 Contents of the "jakarta.inject" Package

The release was published in December 2009 and included in Java EE 6. The fact that Spring doesn't use the annotations by default is a bit strange because Rod Johnson himself initiated the standard.

The Spring documentation doesn't advise using the JSR 330 annotations because Spring's own annotations are more expressive.

2.9.1 Dependency for the JSR 330 Standard Annotation

In the Spring environment, the annotation and the interface can be used. There could be two reasons for this:

- An existing Java/Jakarta EE project will be converted to Spring, and you don't want to modify the annotations via search and replace, but the standard annotations will be recognized.
- A component will be used simultaneously in a Jakarta EE application and in a Spring application.

If you want to use the JSR 330 data types, you have to include a dependency because, unlike the Jakarta standard annotations (e.g., `@PostConstruct`), these injection annotations aren't included in the classpath by default.

To use annotations from JSR 330, include the following in the POM file:

```
<dependency>
  <groupId>jakarta.inject</groupId>
  <artifactId>jakarta.inject-api</artifactId>
  <version>2.0.1</version>
</dependency>
```

2.9.2 Map JSR 330 Annotations to Spring

If the annotations are included via dependency, Spring evaluates them automatically without any additional setting. The mappings are as follows:

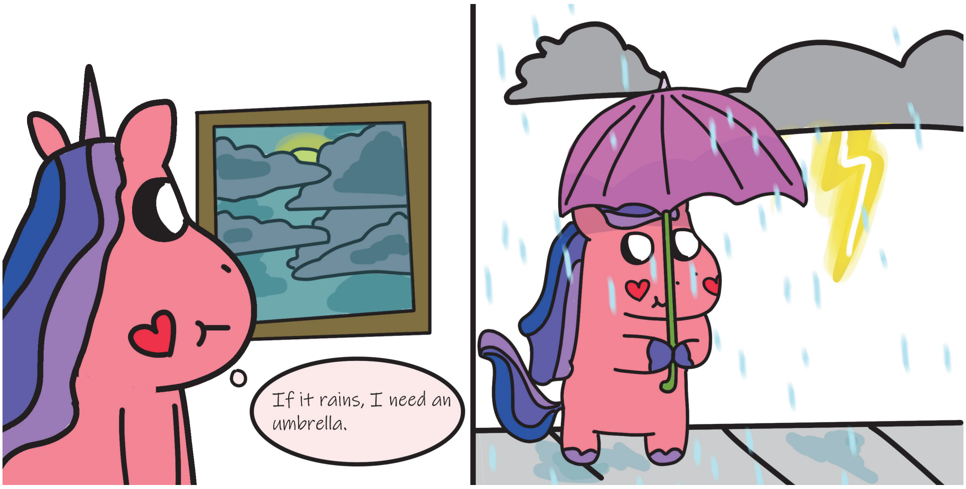
- `@Inject` is like `@Autowired`.
- `@Named` is like `@Component`.

Interestingly, there is another annotation called `@ManagedBean` from JSR 250, *Common Annotations for the Java Platform*. It too is interpreted like an `@Component`.

Because Spring annotations can be used to express much more, it makes no sense for pure Spring applications to rely on the JSR 330 types. For example, `@Inject` doesn't know `required`, and JSR 330 doesn't know `@Value` or `@Lazy` annotations.

2.10 Auto-Configuration

In this section, we discuss the topic that makes Spring Boot so fascinating: *auto-configuration*.


If it rains, I need an umbrella.

2.10.1 @Conditional and Condition

In advance, we'll look at two data types, namely the annotation type `@Conditional` and the interface `Condition`.[30] The special feature is that you can create beans depending on certain conditions, either via `@Component` or via `@Bean` methods. The operation that creates a bean implements the `Condition` interface, and `Condition` has a `matches(…)` method that returns a truth value. Here's afictional example:

```
public class MyCondition implements Condition {
  @Override public boolean matches( ConditionContext context,
                                    AnnotatedTypeMetadata metadata) {
    return …
  }
}
```

The `MyCondition` implementation is set via the `@Conditional` annotation[31] to the `@Bean` method or to the `@Component` annotated class, like this:

- `@Bean @Conditional( MyCondition.class )`
- `@Component @Conditional( MyCondition.class )`

If `@Conditional` is placed at these locations, the Spring bean is generated only if the underlying `condition` is true. This allows us to generate beans at runtime based on the dependency of conditions. For example, we can test whether the operating system is Windows or Linux, whether there is a database connection, whether there is an Internet connection, whether the device has a vibration sensor, and so on. We can basically ask for anything and use that as a conditional on `@Conditional`. Because normally beans are created only once, the condition is generally evaluated only once as well.

Program LowDiskSpaceCondition

Suppose we want to program a condition that returns `true` if there is too little memory to start the program:

```
import org.springframework.context.annotation.*;
import org.springframework.core.type.AnnotatedTypeMetadata;
import org.springframework.util.unit.DataSize;
import java.io.File;

public class LowDiskSpaceCondition implements Condition {

  public boolean matches( ConditionContext __,
```

---

[30] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/context/annotation/Condition.html*

[31] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/context/annotation/Conditional.html*

```
             AnnotatedTypeMetadata ___ ) {
    return DataSize.ofBytes(new File("C:/").getFreeSpace() )
                   .toGigabytes() < 10;
  }
}
```

If the checked device has fewer than 10 gigabytes of free memory, then the `matches(…)` condition returns `true`. Via the parameter list, we get `ConditionContext` and `AnnotatedTypeMetadata`, but we don't have to work with them. The variables are important if we want to access other beans in the container or `Environment` information, that is, configuration properties.

We can put this class, which implements `@Condition`, to a component definition:

```
@Component
@Conditional( LowDiskSpaceCondition.class )
class TempCleaner {

  private final Logger log = LoggerFactory.getLogger( getClass() );

  TempCleaner() {
    log.info( "Cleaning temp directory to acquire more free disk space" );
  }
}
```

For `Conditional`, the type token is specified. `TempCleaner` could, if little memory is available, perhaps delete the *temp* directory first.

When we use an `@Conditional` annotation and specify the type token, this is basically two pieces of information. Therefore, we can combine the two pieces of information into a new annotation type:

```
import org.springframework.context.annotation.Conditional;
import java.lang.annotation.*;

@Retention( RetentionPolicy.RUNTIME )
@Target( { ElementType.TYPE, ElementType.METHOD } )
@Documented
@Conditional( LowDiskSpaceCondition.class )
public @interface ConditionalOnLowDiskSpace { }
```

Later, the custom annotation `ConditionalOnLowDiskSpace` can be used:

```
@ConditionalOnLowDiskSpace
class TempCleaner { … }
```

It looks nicer. This is often done in Spring applications, creating new annotation types that do nothing but cleverly group other annotations together.

Our `@ConditionalOnLowDiskSpace` is a rather special annotation. In practice, there are often similar checks such as the following: Does a property exist, does a bean exist, or what Java version is running? That's why Spring Boot has several default annotation types, so we don't have to implement so many `@Condition` implementations ourselves.

2.10.2 If, Then: @ConditionalOn*

Spring Boot adds a set of predefined `@ConditionalOn*` annotations to the Spring Framework:

- **`@ConditionalOnProperty`**
  This tests if a certain property is set or has a certain value. Then, the match is valid, and a new bean is created.
- **`@ConditionalOnBean`**, **`@ConditionalOnMissingBean`**
  If a bean of a certain type exists, then another bean can be generated. The other way round also works `@ConditionalOnMissingBean`: if a certain bean is missing, then the match becomes valid, and we can generate our own bean.
- **`@ConditionalOnSingleCandidate`**
  There could be multiple Spring-managed beans of a desired type. `@ConditionalOnSingleCandidate` becomes active if there is exactly one candidate for a type.
- **`@ConditionalOnClass`**, **`@ConditionalOnMissingClass`**
  If a certain class is in the classpath, then a bean is created. The opposite also exists: if a certain class is missing in the classpath, then a bean can be created.
- **`@ConditionalOnExpression`**
  This allows us to write any expression of the *Spring Expression Language* (SpEL). And if this expression is `true`, a Spring-managed bean is created.
- **`@ConditionalOnJava`**
  This allows us to query Java versions. For example, if *sealed classes* are included in Java 17 with special help types, this can be expressed with `@ConditionalOnJava`.
- **`@ConditionalOnJndi`**
  This allows us to find out whether certain *Java Naming and Directory Interface* (JDNI) resources are available. JNDI is an access method to extract components (especially from Java Enterprise containers).
- **`@ConditionalOnWebApplication`**, **`@ConditionalOnNotWebApplication`**
  This checks if we're in a web application.
- **`@ConditionalOnResource`**
  This checks if a certain resource is available.
- **`@ConditionalOnCloudPlatform`**
  This allows us to ask if the application is running on a cloud platform. There is an

enumeration type `CloudPlatform`[32] with constants such as `AZURE_APP_SERVICE`, `CLOUD_FOUNDRY`, `HEROKU`, and more.

- **`@ConditionalOnWarDeployment`**
  This checks if we have a web application resource (WAR) file deployment for a servlet container, for example. If the application runs in the embedded server, this condition will return `false`.

These types come from package `org.springframework.boot.autoconfigure.condition`.[33] There are other auto-configurations from other Spring packages, for example, `@ConditionalOnAvailableEndpoint`, or a condition if default security is present, or if metrics should be exported, and so on. These conditions are quite specific, and the Javadoc gives more information.

ConditionalOnClass, ConditionalOnMissingBean, ConditionalOnProperty

Three annotation types are central to Spring Boot, namely `@ConditionalOnClass`, `@ConditionalOnMissingBean`, and `@ConditionalOnProperty`. Basically, you could say that the entire Spring Boot is based on these three annotations. Many things in Spring Boot are determined by external configurations. That is, depending on the configuration, something should happen. With `@ConditionalOnProperty`, properties can be checked to see if they are set and do something based on that. A good example is the connection information to a database via the Java Database Connectivity (JDBC) URL, username, and password. If this information is given, it's probably because the target is a connection to a relational database—and Spring Boot thinks so too! The moment this JDBC connection information exists, for example, a `DataSource` is built, proxy objects are generated, and so on.

Spring Boot is designed to be flexible and accommodating. It doesn't interfere with our work and always allows us to take the lead. Essentially, Spring Boot operates on the principle of "If you build that component, then that's the greatest component in the world, so I don't have to do that. But if you don't build that component, then I will because we'll need that component again later." This is exemplified in the use of `@ConditionalOnMissingBean`. In modern enterprise applications, a vast array of components is required to make anything function properly. Setting up the infrastructure requires configuring an incredible number of things, many of which are preconfigured by Spring Boot.

In addition, `@ConditionalOnClass` is very important for Spring Boot because it provides the magic. The moment certain types are in the classpath, they aren't there for fun, they are performing a task. For example, if Tomcat is in the classpath as a servlet container, it's probably because the application wants to use a web server. In other words, this information about the existence of certain types in the classpath is used to start a server and create objects, among other things.

---

[32] *https://docs.spring.io/spring-boot/docs/current/api/org/springframework/boot/cloud/CloudPlatform.html*

[33] *https://docs.spring.io/spring-boot/docs/current/api/org/springframework/boot/autoconfigure/condition/package-summary.html*

If, Then: @ConditionalOnProperty

Here is an example that demonstrates the usage of `@ConditionalOnProperty` on a factory method:

```
@Bean
@ConditionalOnProperty( name="user.name", havingValue="Christian" )
public Bean create() { … }
```

It's no different with a component:

```
@Component
@ConditionalOnProperty( name="user.name", havingValue="Christian" )
public class ChrisIsThere { … }
```

The condition here is that the username has to be `Christian`. Here, `name` specifies the configuration property, and `havingValue` specifies the value that the property should have. So, this bean will be created only if the username is `Christian`.

ConditionalOnMissingBean

The `@ConditionalOnMissingBean`[34] annotation type is used to ensure that a factory method is invoked or a component annotated with `@Component` is created when a bean of a particular type is absent. Here are some further details:

```
package org.springframework.boot.autoconfigure.condition;

import …

@Target({ ElementType.TYPE, ElementType.METHOD })
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Conditional(OnBeanCondition.class)
public @interface ConditionalOnMissingBean {

    Class<?>[] value() default {};
    String[] type() default {};
    Class<?>[] ignored() default {};
    String[] ignoredType() default {};
    Class<? extends Annotation>[] annotation() default {};
```

---

[34] *https://docs.spring.io/spring-boot/docs/current/api/org/springframework/boot/autoconfigure/condition/ConditionalOnMissingBean.html*

```
    String[] name() default {};
    SearchStrategy search() default SearchStrategy.ALL;
    Class<?>[] parameterizedContainer() default {};
}
```

The details of which missing bean to check for can be set using an array of `Class` objects or an array of fully qualified type names as a string.

Here's a specific example of how `@ConditionalOnMissingBean` is used in the configuration of the Spring Shell project:

```
@Configuration(proxyBeanMethods = false)
public class JLineShellAutoConfiguration {
  …
  @Bean
  @ConditionalOnMissingBean(PromptProvider.class)
  public PromptProvider promptProvider() {
     return () -> new AttributedString("shell:>",
            AttributedStyle.DEFAULT.foreground(AttributedStyle.YELLOW));
    }
}
```

https://github.com/spring-projects/spring-shell/blob/main/spring-shell-autoconfigure/src/main/java/org/springframework/shell/boot/JLineShellAutoConfiguration.java

The class itself is a configuration because it contains only `@Bean` methods. `promptProvider()` is annotated with `@ConditionalOnMissingBean`. This means that if there isn't already an instance of type `PromptProvider`, then this auto-configuration will jump in and build a `PromptProvider` itself. In Section 2.6.2, when we different `PromptProviders` depending on a condition in an `@Bean` method, we built a `PromptProvider` and the default `PromptProvider` disappeared, and now we know the reason.

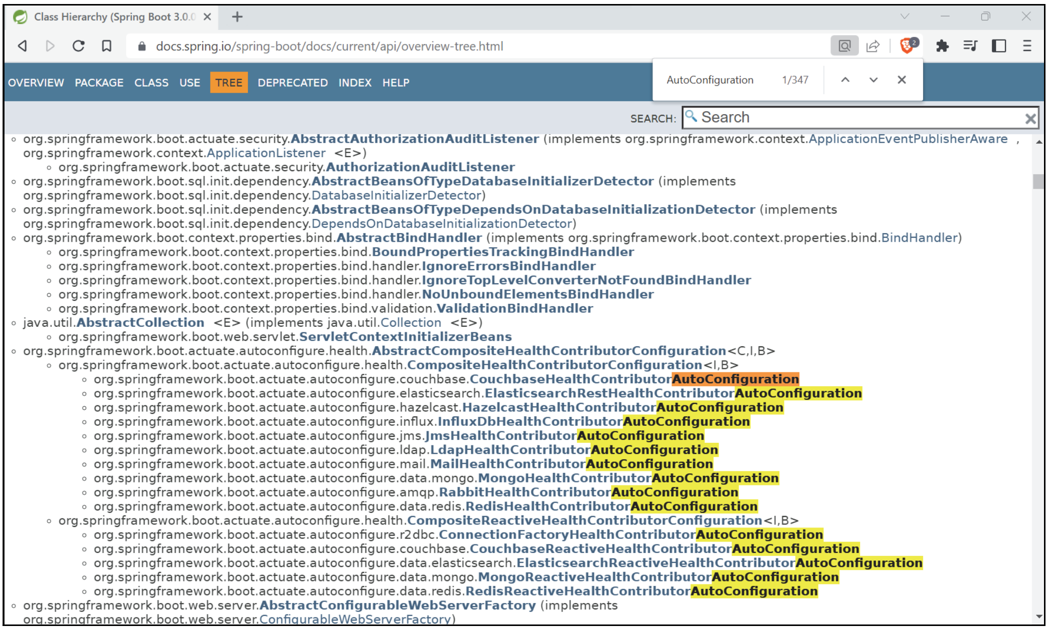
This mechanism, namely that something happens due to certain conditions, is used by Spring Boot in many places. We need to take a closer look at it!

Auto-magically

Much of what Spring does is somehow "magic," except, of course, magic is something that doesn't really exist; magic is just a well-done trick. At its core, Spring Boot's magic is based on this auto-configuration. Fields of application include the following: if a JDBC URL is set, then a `DataSource` is built; if Tomcat is in the classpath, a web server is started and certain security filter chains are set; and so on. This is one of the key differences between Spring Boot and the Spring Framework.

*Auto-Configuration Classes in Spring Boot

The way Spring Boot works is based on `@ConditionalOn*` annotations. In the API documentation,[35] you can see this in the class names (see Figure 2.16).


Class Hierarchy (Spring Boot 3.0.0
docs.spring.io/spring-boot/docs/current/api/overview-tree.html
OVERVIEW
PACKAGE
CLASS
USE
TREE
DEPRECATED
INDEX
HELP
AutoConfiguration
1/347
SEARCH:
Search

- org.springframework.boot.actuate.security.**AbstractAuthorizationAuditListener** (implements org.springframework.context.ApplicationEventPublisherAware , org.springframework.context.ApplicationListener <E>)
    - org.springframework.boot.actuate.security.**AuthorizationAuditListener**
- org.springframework.boot.sql.init.dependency.**AbstractBeansOfTypeDatabaseInitializerDetector** (implements org.springframework.boot.sql.init.dependency.DatabaseInitializerDetector)
- org.springframework.boot.sql.init.dependency.**AbstractBeansOfTypeDependsOnDatabaseInitializationDetector** (implements org.springframework.boot.sql.init.dependency.DependsOnDatabaseInitializationDetector)
- org.springframework.boot.context.properties.bind.**AbstractBindHandler** (implements org.springframework.boot.context.properties.bind.BindHandler)
    - org.springframework.boot.context.properties.bind.**BoundPropertiesTrackingBindHandler**
    - org.springframework.boot.context.properties.bind.handler.**IgnoreErrorsBindHandler**
    - org.springframework.boot.context.properties.bind.handler.**IgnoreTopLevelConverterNotFoundBindHandler**
    - org.springframework.boot.context.properties.bind.handler.**NoUnboundElementsBindHandler**
    - org.springframework.boot.context.properties.bind.validation.**ValidationBindHandler**
- java.util.**AbstractCollection** <E> (implements java.util.Collection <E>)
    - org.springframework.boot.web.servlet.**ServletContextInitializerBeans**
- org.springframework.boot.actuate.autoconfigure.health.**AbstractCompositeHealthContributorConfiguration**<C,I,B>
    - org.springframework.boot.actuate.autoconfigure.health.**CompositeHealthContributorConfiguration**<I,B>
        - org.springframework.boot.actuate.autoconfigure.couchbase.**CouchbaseHealthContributorAutoConfiguration**
        - org.springframework.boot.actuate.autoconfigure.elasticsearch.**ElasticsearchRestHealthContributorAutoConfiguration**
        - org.springframework.boot.actuate.autoconfigure.hazelcast.**HazelcastHealthContributorAutoConfiguration**
        - org.springframework.boot.actuate.autoconfigure.influx.**InfluxDbHealthContributorAutoConfiguration**
        - org.springframework.boot.actuate.autoconfigure.jms.**JmsHealthContributorAutoConfiguration**
        - org.springframework.boot.actuate.autoconfigure.ldap.**LdapHealthContributorAutoConfiguration**
        - org.springframework.boot.actuate.autoconfigure.mail.**MailHealthContributorAutoConfiguration**
        - org.springframework.boot.actuate.autoconfigure.data.mongo.**MongoHealthContributorAutoConfiguration**
        - org.springframework.boot.actuate.autoconfigure.amqp.**RabbitHealthContributorAutoConfiguration**
        - org.springframework.boot.actuate.autoconfigure.data.redis.**RedisHealthContributorAutoConfiguration**
    - org.springframework.boot.actuate.autoconfigure.health.**CompositeReactiveHealthContributorConfiguration**<I,B>
        - org.springframework.boot.actuate.autoconfigure.r2dbc.**ConnectionFactoryHealthContributorAutoConfiguration**
        - org.springframework.boot.actuate.autoconfigure.couchbase.**CouchbaseReactiveHealthContributorAutoConfiguration**
        - org.springframework.boot.actuate.autoconfigure.data.elasticsearch.**ElasticsearchReactiveHealthContributorAutoConfiguration**
        - org.springframework.boot.actuate.autoconfigure.data.mongo.**MongoReactiveHealthContributorAutoConfiguration**
        - org.springframework.boot.actuate.autoconfigure.data.redis.**RedisReactiveHealthContributorAutoConfiguration**
- org.springframework.boot.web.server.**AbstractConfigurableWebServerFactory** (implements org.springframework.boot.web.server.ConfigurableWebServerFactory)

Figure 2.16 Some of the More Than 350 "*AutoConfiguration" Classes

Example MongoAutoConfiguration

We can examine this concept using the example of a concrete auto-configuration such as `MongoAutoConfiguration`.[36] For this, we don't even have to look into the source code, the Javadoc already provides us with this information:

```
@AutoConfiguration ❶
@ConditionalOnClass( MongoClient.class ) ❷
@EnableConfigurationProperties( MongoProperties.class )
@ConditionalOnMissingBean(
  "org.springframework.data.mongodb.MongoDatabaseFactory"
)
public class MongoAutoConfiguration { ❸
  @Bean
  @ConditionalOnMissingBean( MongoClient.class )  ❹
  public MongoClient mongo(…) { … }
}
```

---

[35] *https://docs.spring.io/spring-boot/docs/current/api/*

[36] *https://docs.spring.io/spring-boot/docs/current/api/org/springframework/boot/autoconfigure/mongo/MongoAutoConfiguration.html*

❶ Spring Boot annotated all auto-configuration classes with `@AutoConfiguration`.[37]

❷ The auto-configuration should run only if type `MongoClient` is present in the classpath. This is exactly what `@ConditionalOnClass` checks. If class `MongoClient` is present, then this rule takes effect, and it goes on.

❸ The class is `public`, but we shouldn't overstate that. We shouldn't interfere with these visible data types because they are none of our business. For example, if any configurations or similar things are declared, then that isn't for us.

❹ If class `MongoClient` exists, then the next test follows because just the class in the classpath alone isn't enough for Spring Boot to build an instance of a bean. That's why there's a second `@ConditionalOnMissingBean` that says, "If there's no instance of `MongoClient` yet, auto-configuration will build one." This is a typical `@Bean` method that we have as a factory method.

The code nicely shows the cascading of `@ConditionalOnClass` on the component and of `@ConditionalOnMissingBean` on a factory method.

[...]AutoConfiguration.imports References *AutoConfiguration Classes

The auto-configurations are regular components but aren't detected via classpath scanning. Custom auto-configurations may very well be part of classpath scanning, but Spring Boot uses a slightly different technique because the Spring Boot Starter has a dependency on `spring-boot-autoconfigure`. There is a special *META-INF* directory there that has subdirectory *spring* and file *org.springframework.boot.autoconfigure.AutoConfiguration.imports*.[38] (This long file name results from the fully qualified type name of the annotation.)

This file contains lines with auto-configurations; let's look inside the file:

```
org.springframework.boot.autoconfigure.admin.
SpringApplicationAdminJmxAutoConfiguration
org.springframework.boot.autoconfigure.aop.AopAutoConfiguration
org.springframework.boot.autoconfigure.amqp.RabbitAutoConfiguration
org.springframework.boot.autoconfigure.batch.BatchAutoConfiguration
org.springframework.boot.autoconfigure.cache.CacheAutoConfiguration
org.springframework.boot.autoconfigure.cassandra.CassandraAutoConfiguration
org.springframework.boot.autoconfigure.context.
ConfigurationPropertiesAutoConfiguration
```

---

[37] In previous source code, it was still a normal `@Configuration`; `@AutoConfiguration` is new since Spring Boot 2.7.

[38] Prior to Spring Boot 2.7, this information was housed slightly differently. There may also be a *spring.factories* file in the *META-INF* directory, and it used to list autoconfigurations. It's still evaluated for compatibility reasons. However, *spring.factory* contains an entire set of other configuration properties, so the new way is cleaner.

```
…
org.springframework.boot.autoconfigure.mongo.MongoAutoConfiguration
…
```

Listing 2.21 https://github.com/spring-projects/spring-boot/blob/main/spring-boot-project/spring-boot-autoconfigure/src/main/resources/META-INF/spring/org.springframework.boot.autoconfigure.AutoConfiguration.imports

The file has about 140 lines. `MongoAutoConfiguration` also appears.

So, these auto-configurations aren't detected via classpath scanning. These are internal Spring Boot classes and are recognized because they are in this file. When Spring Boot starts, this file is opened and processed in order. Spring Boot goes through each auto-configuration, and then new components may be created.

That's also what a custom starter implementation with auto-configurations would look like because the idea of a starter is to preconfigure certain things and not get in our way if we were to build these components ourselves.

@AutoConfiguration Annotation Type

The `@AutoConfiguration` annotation is applied to the auto-configuration classes. Here's an example declaration:

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Configuration(proxyBeanMethods = false)
@AutoConfigureBefore @AutoConfigureAfter
public @interface AutoConfiguration {
  @AliasFor(annotation = Configuration.class)
  String value() default "";

  @AliasFor(annotation = AutoConfigureBefore.class, attribute = "value")
  Class<?>[] before() default {};

  @AliasFor(annotation = AutoConfigureBefore.class, attribute = "name")
  String[] beforeName() default {};

  @AliasFor(annotation = AutoConfigureAfter.class, attribute = "value")
  Class<?>[] after() default {};

  @AliasFor(annotation = AutoConfigureAfter.class, attribute = "name")
  String[] afterName() default {};
}
```

Basically, an `@AutoConfiguration` is a regular `@Configuration` with the restriction that `proxyBeanMethods` equals `false` by default. This means that Spring won't build a subtype.

Essentially, the annotation type allows you to build a dependency chain which can express that an auto-configuration should be executed and initialized before or after another auto-configuration. The Javadoc of the annotation type[39] talks about these auto-configuration classes being loaded by a `SpringFactoriesLoader`—and that's exactly what we just looked at with these particular files where the auto-configuration classes are entered fully qualified.

And Connections of Auto-Configuration

Let's revisit the auto-configuration of `MongoClient`:

```
@AutoConfiguration
@ConditionalOnClass( MongoClient.class )
@EnableConfigurationProperties( MongoProperties.class )
@ConditionalOnMissingBean(
  "org.springframework.data.mongodb.MongoDatabaseFactory"
)
```

The code shows `@ConditionalOnClass` and `@ConditionalOnMissingBean`. Two conditions mean that both must be true at the same time. That is, `MongoClient` must be in the classpath as a class, and, on the other hand, a bean of type `MongoDatabaseFactory` must not already exist. Only when both conditions are true is the auto-configuration class considered at all.

2.10.3 Turn on Spring Debug Logging

One problem in practice is that it's not necessarily clear which auto-configurations have been activated at all and which haven't. That's why there is a `debug` configuration property that can be set. There are different ways to set this property:

- On the command line: `--debug` or `-Ddebug`
- In code: `SpringApplication.run(*.class, "--debug");` (hack)
- In *application.properties*: `debug=true`
- In *application.yaml*: `debug: true`

If the property is set, there is a long output with several hundred lines.

The output starts with a `Conditions Evaluation Report` as the headline and then `Positive matches` follow:

---

[39] *https://docs.spring.io/spring-boot/docs/current/api/org/springframework/boot/autoconfigure/AutoConfiguration.html*

```
============================
CONDITIONS EVALUATION REPORT
============================

Positive matches:
-----------------

 AopAutoConfiguration matched:
  - @ConditionalOnProperty (spring.aop.auto=true) matched (OnPropertyCondition)
…
 ShellRunnerAutoConfiguration#interactiveApplicationRunner matched:
  - @ConditionalOnProperty (spring.shell.interactive.enabled=
true) matched (OnPropertyCondition)
…
```

The positive matches list which conditions were true, so based on that, certain Java beans are created. For example, we see the JShell. That is because the `spring.shell.interactive.enable` switch is `true` (or rather, we didn't set it, it was the default), Spring Boot automatically created a Spring-managed bean.

After the positive matches come the `negative matches`:

```
Negative matches:
-----------------

 AopAutoConfiguration.AspectJAutoProxyingConfiguration:
  Did not match:
   - @ConditionalOnClass did not find required class
 'org.aspectj.weaver.Advice' (OnClassCondition)
…
 MongoAutoConfiguration:
  Did not match:
   - @ConditionalOnClass did not find required class
 'com.mongodb.client.MongoClient' (OnClassCondition)
…
```

Shown here are the things that don't match, and these are technologies that we don't use in our code at all, like access to the MongoDB database. This is correct because `ConditionalOnClass` only matches if the `MongoClient` class is in the classpath.

Finally, there is an `Exclusions` item (auto-configurations that we've excluded, which are none by default) and an `Unconditional classes` item (auto-configurations that are always applied):

```
Exclusions:
-----------

  None


Unconditional classes:
----------------------

  org.springframework.shell.boot.SpringShellAutoConfiguration
  org.springframework.boot.autoconfigure.context
.ConfigurationPropertiesAutoConfiguration
  org.springframework.shell.boot.ExitCodeAutoConfiguration
  org.springframework.shell.boot.ThemingAutoConfiguration
  …
  org.springframework.shell.boot.JLineAutoConfiguration
  org.springframework.shell.boot.StandardAPIAutoConfiguration
  org.springframework.boot.autoconfigure.info.ProjectInfoAutoConfiguration
```

Note

Enabling debug logging doesn't mean that all logging messages will be displayed more urgently than `DEBUG`!

2.10.4 Controlling Auto-Configurations Individually *

The reason Spring Boot takes auto-configurations into account is because of the annotation that is set as a meta-annotation at `@SpringBootApplication`:

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@SpringBootConfiguration
@EnableAutoConfiguration
@ComponentScan( … )
public @interface SpringBootApplication {
  …
}
```

`@SpringBootApplication` has the `@EnableAutoConfiguration` meta-annotation, and this causes Spring Boot to run through the auto-configurations on its own. This has nothing to do with taking into account custom classes annotated with `@Configuration` (they are taken into account, of course), but only with Spring Boot processing the predefined auto-configurations.

Exclude Auto-Configurations

Not all auto-configurations are always desired. In the end, an unused auto-configuration causes higher startup time. In addition, an auto-configuration may lead to certain data types because something could be misinterpreted, but these objects aren't needed and may even create conflicts.

Therefore, there are several ways to exclude auto-configurations or to manually control the whole configuration. Explicitly, auto-configurations can be excluded via the annotation attribute exclude at `@EnableAutoConfiguration`. This looks like the following:

```
@EnableAutoConfiguration( exclude = {
  DataSourceAutoConfiguration.class,
  DataSourceTransactionManagerAutoConfiguration.class,
  HibernateJpaAutoConfiguration.class
} )
```

The `exclude` attribute contains an array of `Class` objects.

The second option is provided by `spring.autoconfigure.exclude`, a configuration property that contains an array of specifications. If we take the *application.properties* file, the two ways to set multiple elements are

```
spring.autoconfigure.exclude= ↩
org.springframework.boot.autoconfigure.jdbc.DataSourceAutoConfiguration, ↩
DataSourceTransactionManagerAutoConfiguration.class, ↩
org.springframework.boot.autoconfigure.orm.jpa.HibernateJpaAutoConfiguration
```

or

```
spring.autoconfigure.exclude[0]= ↩
[…].autoconfigure.jdbc.DataSourceAutoConfiguration
spring.autoconfigure.exclude[1]= ↩
[…].autoconfigure.jdbc.DataSourceTransactionManagerAutoConfiguration
spring.autoconfigure.exclude[2]= ↩
[…].autoconfigure.orm.jpa.HibernateJpaAutoConfiguration
```

In the first variant, the fully qualified class names are written comma-separated one after the other; in the second variant, the entries are determined with an index.

In summary, to exclude auto-configuration in Spring Boot, we have the following choices:

- Control this declaratively via annotation attribute `exclude` using annotation `@EnableAutoConfiguration`.
- Disable this via an external configuration, for example, via the *application.properties* file.

Turn Off All Auto-Configurations

The `@SpringBootApplication` annotation is, at its core, nothing more than a conjunction of `@SpringBootConfiguration`, `@EnableAutoConfiguration`, and `@ComponentScan`.

Replacing `@SpringBootApplication` with `@SpringBootConfiguration` and `@ComponentScan` (and omitting `@EnableAutoConfiguration`) leaves a tiny number of beans:

```
date4uApplication
org.springframework.boot.autoconfigure.internalCachingMetadataReaderFactory
org.springframework.context.annotation.internalAutowiredAnnotationProcessor
org.springframework.context.annotation.internalCommonAnnotationProcessor
org.springframework.context.annotation.internalConfigurationAnnotationProcessor
org.springframework.context.event.internalEventListenerFactory
org.springframework.context.event.internalEventListenerProcessor
```

In other words, all the auto-configurations are gone. Now you can start adding targeted auto-configurations back.

Individual Auto-Configurations

The `@Import` annotation can be used to specify an array of type tokens, and this can be used to specify exactly the auto-configurations that you would like to apply. This is how it might look:

```
@Configuration
@Import( {
    PropertyPlaceholderAutoConfiguration.class,
    DispatcherServletAutoConfiguration.class,
    EmbeddedServletContainerAutoConfiguration.class,
    ServerPropertiesAutoConfiguration.class,
    ErrorMvcAutoConfiguration.class,
    HttpEncodingAutoConfiguration.class,
    HttpMessageConvertersAutoConfiguration.class,
    WebMvcAutoConfiguration.class,
    JacksonAutoConfiguration.class
} )
public class WebConfiguration { }
```

The advantage is that you can optimize the startup time by a few milliseconds—because when Spring Boot has to test through more than 300 auto-configurations, it takes a little while.

Whether you want to go that far or not is a matter of taste. The vast majority of applications deliberately use Spring Boot and the auto-configurations because it saves a lot of work and makes life easier. However, if you're interested in reducing the startup time to an absolute minimum, the individual auto-configuration is the way to go.

2.11 Spring Expression Language

In Spring, the *Spring Expression Language (SpEL)* provides a powerful syntax for querying and manipulating objects at runtime. SpEL enables developers to perform operations such as checking states, querying object graphs, and causing side effects declaratively, using expressions that are written in a special syntax.

While these operations are traditionally expressed in code, there are situations where it's more appropriate to express them declaratively, such as in annotations or XML files. With SpEL, developers can set statements declaratively, which are then executed at runtime.

SpEL is a core part of the Spring Framework, and it offers a variety of features, such as the ability to reference properties, call methods, and perform arithmetic operations. SpEL also supports conditional expressions and regex, making it a versatile tool for developers.

The expressions that are written with SpEL are called *SpEL expressions*. To gain a better understanding of SpEL, it's useful to explore some examples and learn the basic syntax.

2.11.1 ExpressionParser

An `ExpressionParser`[40] can be used to evaluate SpEL expressions. An object can be created via `SpelExpressionParser`[41] via the constructor:

```
ExpressionParser parser = new SpelExpressionParser();
```

The `ExpressionParser` type has method `parseExpression(String)` that evaluates a SpEL.

Here are a few examples:

```
Expression exp = parser.parseExpression( "1+2*3 >= 7 and false" );
System.out.println( exp.getValue() ); // false
```

As you can see, you can evaluate arithmetic expressions, including the precedence, of course, so that `1 + 2 * 3` is `7`. Comparison operators exist as well. For better readability, some operators have words such as `or`, `and`, or `not`. The overall result of the shown expression is `false`.

```
exp = parser.parseExpression( "'Hello '.concat('World').toUpperCase()" );
System.out.println( exp.getValue() ); // HELLO WORLD
```

Strings are written in Java with double quotes, and because SpEL can contain strings, single quotes are used to specify characters or strings. Regular methods can be called on the `String` object, for example, `concat(…)` or `toUpperCase()`.

---

[40] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/expression/ExpressionParser.html*

[41] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/expression/spel/standard/SpelExpressionParser.html*

```
exp = parser.parseExpression( "new java.awt.Point(10,20).location.x" );
System.out.println( exp.getValue() ); // 10.0
```

In a SpEL expression, objects can be built with the keyword `new`. Property accesses are possible, that is, `location` and `x` aren't variables, but the SpEL engine automatically calls the getters, that is, first `getLocation()` on the `Point` and then `getX()`. There is a public variable `x` on the `Point` object, but the `getX()` method is still called.

```
exp = parser.parseExpression( "{1,2,3,4}.get(0) == 1 ? 'one' : 'else'" );
System.out.println( exp.getValue() ); // one
```

Anything enclosed in curly braces is a list and translates to a `java.util.List`. `List` has a `get(int)` method; that is, the expression accesses the first element, and that is 1. In SpEL, there is the conditional operator as well.

```
exp = parser.parseExpression( "'12345' matches '\\d+'" );
System.out.println( exp.getValue() ); // true
```

SpEL has a special keyword `matches` that we can use to make simple and straightforward regex queries. Again, we have a string `12345`, and the question is whether this matches any number of digits. It does!

```
exp = parser.parseExpression( "T(java.lang.Math).random()" );
System.out.println( exp.getValue() );
```

The notation seems strange in part because it's written with `T()`. This is used for calling static methods. A fully qualified class name is placed in the round brackets:

```
exp =parser.parseExpression("T(java.lang.Runtime).getRuntime().exec('cmd')");
System.out.println( exp.getValue() );
// Process[pid=12345, exitValue="not exited"]
```

SpEL can be dangerous in principle, and developers must be careful not to open the door to attackers. In the output, you can see that the process was validly started, which means there could be a program running that could ruin our computer. However, certain precautions are taken by the expression parser based on the configuration.

2.11.2 SpEL in the Spring (Boot) API

Spring uses SpEL in many places; as examples, here follows a passage from Spring Boot and two passages from the Spring Framework itself:

- `@ConditionalOnExpression`
- `@Cacheable`
- `@Scheduled`

@ConditionalOnExpression

At `@ConditionalOnExpression`,[42] we can register a managed bean depending on a condition. The condition is specified as an annotation attribute. The declaration of the type looks like this:

```
@Retention(RetentionPolicy.RUNTIME)
@Target({ ElementType.TYPE, ElementType.METHOD })
@Documented
@Conditional(OnExpressionCondition.class)
public @interface ConditionalOnExpression {

    /**
     * The SpEL expression to evaluate. Expression should return
     * {@code true} if the condition passes or {@code false} if it fails.
     * @return the SpEL expression
     */
    String value() default "true";

}
```

The Javadoc makes it clear that a new component is included in the context if a SpEL expression returns `true`, and not if it returns `false`.

Let's connect this to a concrete example:

```
@Component
@ConditionalOnExpression( "1+2=3" )
// @ConditionalOnExpression(
//     "not (systemProperties['user.home'] ?:'c:/').isEmpty()"
// )
// @ConditionalOnExpression( "@spEL != null" )
class Renderer { }

@SpringBootApplication
public class SpEL {

  public static void main( String[] args ) {
    ApplicationContext ctx = SpringApplication.run( SpEL.class, args );
    System.out.println( ctx.containsBean( "renderer" ) );  // true
  }
}
```

First, the main program `SpEL` starts the container normally; the class `SpEL` is the main configuration. The program asks with `containsBean(…)` if there is a bean with the name

---

[42] *https://docs.spring.io/spring-boot/docs/current/api/org/springframework/boot/autoconfigure/condition/ConditionalOnExpression.html*

"renderer", and everything above leads to a bean with the name "renderer", so the outputs will always be "true".

```
@ConditionalOnExpression( "1+2=3" )
```

The first test is simple: Is it true that `1 + 2 = 3`? Of course! So, the condition is `true`, and consequently the bean `renderer` is built.

```
@ConditionalOnExpression(
     "not (systemProperties['user.home'] ?:'c:/').isEmpty()"
)
```

The second SpEL expression asks whether the system properties have an assignment for `user.home`, and if so, the result is the assignment of `user.home`. Otherwise, the result is the string `c:`. The test is done by the Elvis operator `? :`. Many programming languages have this operator, but Java doesn't. The Elvis operator is a short form of the condition operator, which expresses that if the expression `systemProperties['user.home']` isn't `null`, it's the result; otherwise, the alternative is `c:`. Any result will be nonempty, so `isEmpty()` will return `false`; that turns `not` around, so it comes out `true` again in the end.

```
@ConditionalOnExpression( "@spEL != null" )
```

The third example shows how to fall back on Spring-managed beans, namely with the `@` sign. The SpEL expression asks if a component named "spEL" is `non-null`. This is true because the class is called `SpEL` and gets the automatically lowercase bean name `spEL`. Basically, you could have called methods on the Spring-managed bean again.

@Cacheable

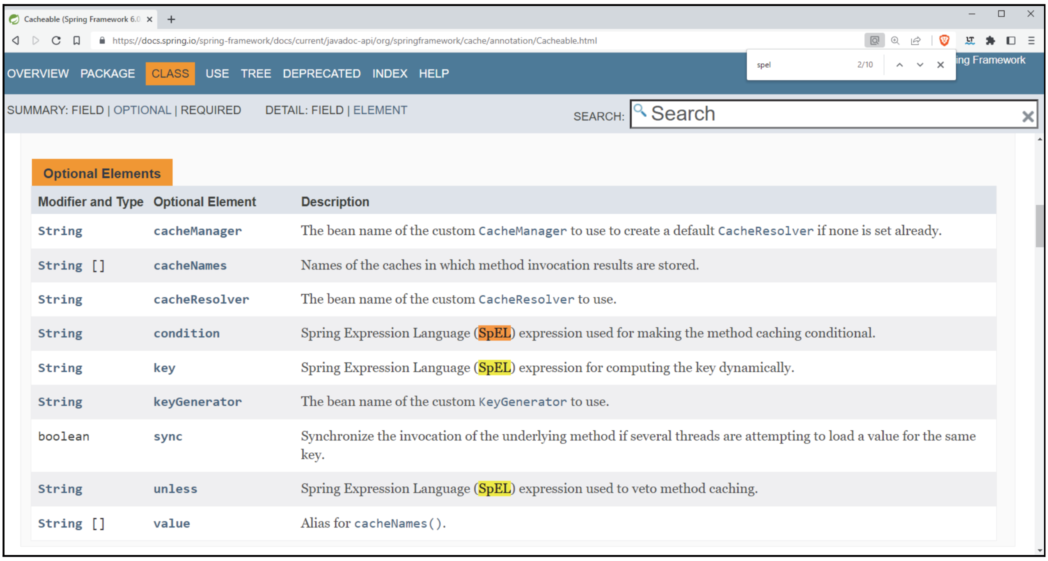
Another example using the `@Cacheable` annotation allows for caching method returns. This means that if a method is called again with the same arguments, the cached value can be returned instead of recalculating it. We'll discuss `@Cacheable` further in Chapter 4, Section 4.2.

With the help of SpEL, various aspects can be controlled declaratively. For example, caching should only occur when certain conditions apply. If you look at the Javadoc,[43] you can easily see the annotation attributes that can be assigned to a SpEL expression (see Figure 2.17).

Using these declarative SpEL expressions, we can refine queries because where else would the conditions be? We would have to program out the cache rules otherwise. Using declarative expressions on the annotations is elegant.

---

[43] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/cache/annotation/Cacheable.html*


Cacheable (Spring Framework 6.0
https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/cache/annotation/Cacheable.html
spel
2/10
ing Framework
OVERVIEW PACKAGE CLASS USE TREE DEPRECATED INDEX HELP
SUMMARY: FIELD | OPTIONAL | REQUIRED DETAIL: FIELD | ELEMENT
SEARCH:
Search

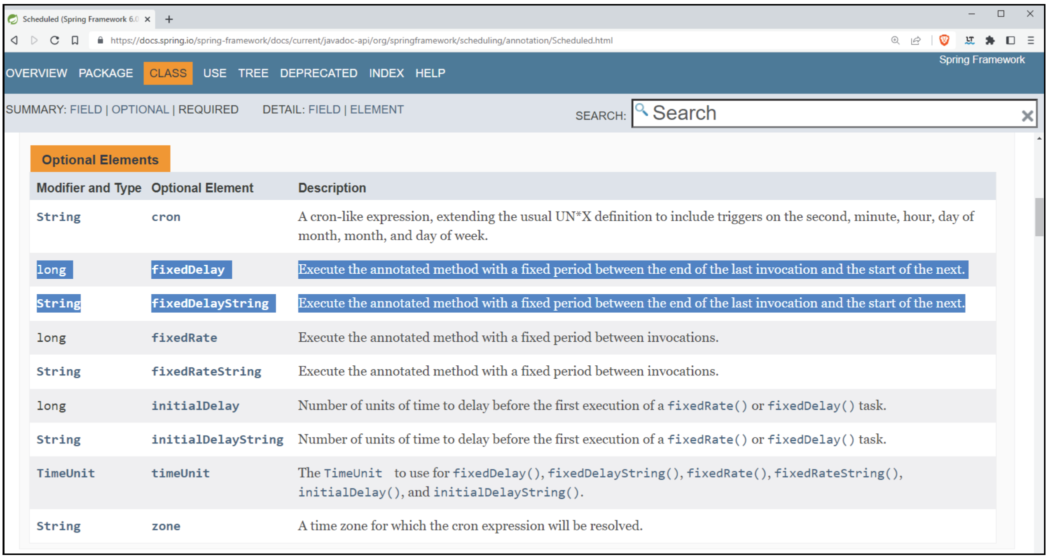
Optional Elements

Modifier and Type	Optional Element	Description
`String`	`cacheManager`	The bean name of the custom `CacheManager` to use to create a default `CacheResolver` if none is set already.
`String []`	`cacheNames`	Names of the caches in which method invocation results are stored.
`String`	`cacheResolver`	The bean name of the custom `CacheResolver` to use.
`String`	`condition`	Spring Expression Language (SpEL) expression used for making the method caching conditional.
`String`	`key`	Spring Expression Language (SpEL) expression for computing the key dynamically.
`String`	`keyGenerator`	The bean name of the custom `KeyGenerator` to use.
`boolean`	`sync`	Synchronize the invocation of the underlying method if several threads are attempting to load a value for the same key.
`String`	`unless`	Spring Expression Language (SpEL) expression used to veto method caching.
`String []`	`value`	Alias for `cacheNames()`.

Figure 2.17 Annotation Attributes for SpEL Expressions

@Scheduled

For a third example, take a look at the annotation `@Scheduled`.[44] Scheduling is about Spring being able to call methods on a regular basis. Now, with `@Scheduled`, there are two ways to specify values: once via a `long` and the other via a SpEL string (see Figure 2.18).


Scheduled (Spring Framework 6.0
https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/scheduling/annotation/Scheduled.html
Spring Framework
OVERVIEW PACKAGE CLASS USE TREE DEPRECATED INDEX HELP
SUMMARY: FIELD | OPTIONAL | REQUIRED DETAIL: FIELD | ELEMENT
SEARCH:
Search

Optional Elements

Modifier and Type	Optional Element	Description
`String`	`cron`	A cron-like expression, extending the usual UN*X definition to include triggers on the second, minute, hour, day of month, month, and day of week.
`long`	`fixedDelay`	Execute the annotated method with a fixed period between the end of the last invocation and the start of the next.
`String`	`fixedDelayString`	Execute the annotated method with a fixed period between the end of the last invocation and the start of the next.
`long`	`fixedRate`	Execute the annotated method with a fixed period between invocations.
`String`	`fixedRateString`	Execute the annotated method with a fixed period between invocations.
`long`	`initialDelay`	Number of units of time to delay before the first execution of a `fixedRate()` or `fixedDelay()` task.
`String`	`initialDelayString`	Number of units of time to delay before the first execution of a `fixedRate()` or `fixedDelay()` task.
`TimeUnit`	`timeUnit`	The `TimeUnit` to use for `fixedDelay()`, `fixedDelayString()`, `fixedRate()`, `fixedRateString()`, `initialDelay()`, and `initialDelayString()`.
`String`	`zone`	A time zone for which the cron expression will be resolved.

Figure 2.18 Alternative Specifications via Numeric Values and SpEL Strings

---

[44] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/scheduling/annotation/Scheduled.html*

That should be enough about SpEL for now. We'll come back to it in a few places, but in everyday life, you won't find SpEL that often. In addition, you have to keep in mind that such expressions are parsed only at runtime, and because they are strings, we have type safety issues, so when refactoring, we must not forget to adjust these strings. Users of *IntelliJ Ultimate Edition* enjoy the advantage that the IDE looks into SpEL expressions and detects certain errors—this is definitely a big bonus.

2.12 Summary

Filling the container with Spring-managed beans is one of our main tasks. There are many small aspects to consider, and Spring still offers some subtleties that haven't been presented in this chapter. If you want to master the full capabilities of the container, you can find all the necessary details in the Spring Framework reference documentation at *https://docs.spring.io/spring-framework/docs/current/reference/html/core.html#spring-core*. XML configuration in particular still plays a dominant role in the reference documentation.

Chapter 3

Selected Modules of the Spring Framework

So far, we've dealt exclusively with building components and wiring. However, Spring can do much more, and, in this chapter, we'll look at a few more capabilities.

3.1 Helper Classes in Spring Framework

The Spring Framework is a vast and complex framework that provides numerous classes to facilitate the development of enterprise-level applications. While some classes are intended for internal use and aren't relevant to developers, others are crucial because they are implemented by developers in their applications. In addition, there are also helper classes that can be leveraged by developers to aid in the development process. As such, it's essential for developers to have a good understanding of the various types of classes provided by the Spring Framework and their respective use cases to develop efficient and effective applications.

3.1.1 Components of org.springframework

The Spring Framework contains about 400 packages with about 4,000 type declarations. The Javadoc at *https://docs.spring.io/spring-framework/docs/current/javadoc-api/overview-summary.html* gives an overview. The content is also extensive because a wide variety of technologies appear in the Javadoc. Besides the core framework, there is the *Spring Web model-view-controller (MVC) framework*, database access, messaging, and so on.

The Spring Framework is internally composed of the following parts:[1]

- `spring-aop`
- `spring-aspects`
- `spring-beans`
- `spring-context`
- `spring-context-indexer`
- `spring-context-support`

---

[1] *https://github.com/spring-projects/spring-framework/blob/main/settings.gradle*

- `spring-core`
- `spring-core-graalvm`
- `spring-core-test`
- `spring-expression`
- `jumping instrument`
- `spring-jcl`
- `spring-jdbc`
- `spring-jms`
- `spring-messaging`
- `spring-orm`
- `spring-oxm`
- `jump-r2dbc`
- `jump-test`
- `spring-tx`
- `spring-web`
- `spring-webflux`
- `spring-webmvc`
- `spring-websocket`
- `framework-bom`
- `integration-tests`

The longer you work with Spring, the more relaxed you become in the knowledge that you don't need to know that much at all, but can still be highly productive.

We'll deal with almost all modules at least superficially in this book, so that you can get a good idea of what you need these modules for and what you can do with them.

Note: Spring versus Other Enterprise Frameworks

In contrast to Jakarta Enterprise Edition (Jakarta EE), where there is a clear separation between an application programming interface (API; the specification) and the implementation provided by an application server, Spring Framework's approach is much more tightly integrated. In Spring, there is no distinction between "container" and "implementation" as both form a unified entity. This is why Spring's Javadoc is extensive, as it contains many internal implementations details that end users might not come into direct contact with.

3.2 External Configuration and the Environment

In this section, we'll deal with the external configuration and special data type `Environment`.

3.2.1 Code and External Configuration

It's considered a best practice to avoid using magic values, such as hard-coded numerical values or path separators in the code. Instead, it's recommended to store them in variables, typically constants. However, it's not always practical to define these constants in the code, especially for file names, server ports, URLs, paths, and other configuration details. This is because these constants might need to be changed often, depending on the environment or system on which the software is running. If these constants are hard-coded in the program, then the program would need to be recompiled each time they were changed, which is time-consuming and inflexible.

To address this issue, it's common to keep certain information external, so that the software can read in the configurations. This way, changes can be made to the configurations without requiring any changes to the code. This approach makes the software more flexible and adaptable to different environments. Common ways to store external configurations include property files, XML files, environment variables, command-line arguments, and database tables. By using external configurations, the software can be easily configured to work on different systems or environments, without requiring any code changes.

Example of Values in the application.properties File

Spring collects configuration information from various sources and places it in an *environment*. The sources from which configuration information comes are called *property sources*. One of these property sources, which we discussed in Chapter 1, Section 1.4, is the *application.properties* text file in the classpath.

We will set two configurations:

```
com.tutego.homepage=http://tutego.com/
com.tutego.number-of-seminars=350
# port=80
```

The first two properties indicate a qualification via a dot, which creates a kind of namespace; while this isn't mandatory, it's a best practice. Namespaces and longer keys collide less often, so `port` isn't a good example. Then, number sign `#`, as we see it for a `port`, is used for comments.

Another advantage of hierarchical properties is that they can be written elegantly in YAML Ain't Markup Language (YAML) files; in Section 3.2.10, we'll take a closer look at YAML files.

Note

The key of a property may get written several times. In such a case, the last write access wins and overwrites what was written before.

It's common to use lowercase for configuration properties and to write composite segments in the *kebab format* for configuration properties. That is, a minus sign separates the corresponding segments.

3.2.2 Environment

All property sources come together conceptually in a large container of type `Environment`.[2] This `Environment` can be thought of as a concatenated list of different `java.util.Properties`, only with more possibilities.

The `Environment` contains all key-value pairs and offers the possibility to group certain properties using profiles; this will be the subject of Section 3.2.11. The Spring Framework provides an implementation of the `Environment` interface that can be injected.

PropertyResolver and Environment

The `Environment` itself has few methods, and they are needed when using profiles. The interesting methods for querying the properties come from the `Environment`'s supertype: `PropertyResolver`[3] (see Figure 3.1). This type declares important methods such as `getProperty(…)` or `containsProperty(…)`.

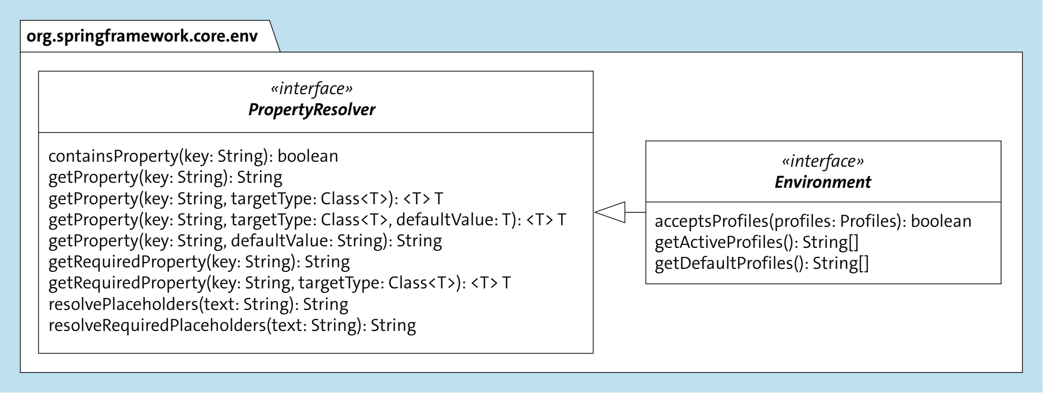

org.springframework.core.env
«interface»
PropertyResolver
containsProperty(key: String): boolean
getProperty(key: String): String
getProperty(key: String, targetType: Class<T>): <T> T
getProperty(key: String, targetType: Class<T>, defaultValue: T): <T> T
getProperty(key: String, defaultValue: String): String
getRequiredProperty(key: String): String
getRequiredProperty(key: String, targetType: Class<T>): <T> T
resolvePlaceholders(text: String): String
resolveRequiredPlaceholders(text: String): String
«interface»
Environment
acceptsProfiles(profiles: Profiles): boolean
getActiveProfiles(): String[]
getDefaultProfiles(): String[]

Figure 3.1 Example for Requesting Properties

---

[2] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/core/env/Environment.html*

[3] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/core/env/PropertyResolver.html*

Here's a small snippet to demonstrate these methods:

```
log.info( env.containsProperty( "user.home" ) );
log.info( env.getProperty( "user.home" ) );
log.info( env.getProperty( "com.tutego.homepage" ) );
log.info( env.getProperty( "com.tutego.number-of-seminars",Integer.class ) );
log.info( env.getProperty( "com.tutego.homepage", "http://tutego.com/" ) );
```

Method `containsProperty(…)` determines whether a certain configuration property exists. The example uses `user.home`, which is a variable from the system properties. `Environment` is filled with several property sources, and one of these property sources is the system properties.

The actual assignment of a property is queried by `getProperty(…)`. The environment variable `user.home` exists because the property comes from the system properties, and the user directory is stored there. In addition, our own property `com.tutego.homepage` is added because it was declared in the *application.properties* file. This file is also a property source and is included in the `Environment`.

All properties are strings per se, but the `getProperty(…)` method is overloaded so that it can also convert types via a `ConversionService`, for example, from a string to an integer. So the property `com.tutego.numbers-of-seminars` is converted to an `Integer`. The possibilities the conversion service gives us are shown in Section 3.6 in more detail.

The last `getProperty(…)` method shows that default values can also be set.

Output the User Directory

To try this out in a real example, we can implement new shell command `user-home` in the `FsCommands` class, which returns the user directory via the `Environment`.

```
@Autowired
private Environment env;

@ShellMethod( "Display user home" )
public String userHome() {
  return env.getProperty( "user.home" );
}
```

Listing 3.1 FsCommands Extension

3.2.3 Inject Values with @Value

The Spring Framework declares an `@Value` annotation,[4] which can be used to inject values into an object. This is similar to `@Autowired`; but while `@Autowired` injects references into Spring-managed beans only, `@Value` can also inject `Environment` values or dynamic computations. There is another similarity: wiring is valid with field injection, setter injection, and constructor injection, and visibility doesn't matter. The same is true for `@Value`.

---

[4] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/beans/factory/annotation/Value.html*

Here are a few examples:

```
@Component
class ClassWithNickname {
  @Value( "FillmoreFat" )
  String nickname;
}
```

The instance variable `nickname` is annotated with `@Value`, and that means when the component is initialized later, `nickname` will be assigned to "FillmoreFat". As it stands here, it's not particularly helpful. We could have written `nickname = "FillmoreFat"` right away, but the trick is that this assignment can come from a dynamic configuration.

For setters, `@Value` is also possible:

```
@Component
class ClassWithActiveProfile {
  @Value( "true" )
  void setActiveProfile( boolean isActiveProfile ) { … }
}
```

The setter receives the `boolean` value `true`. The configuration property is a `String` and is automatically converted to the desired data type `boolean` by the internal converters (a Spring-managed bean of type `ConversionService`).

The annotation `@Value` is placed before the setter but can also be placed at the parameter variable:

```
@Autowired
void setActiveProfile( @Value("true") boolean isActiveProfile )
```

In this case, however, the annotation `@Autowired` must be placed in front of the method because the Spring Framework doesn't inspect `@Value` annotated parameters.

In the third variant, `@Value` can be used at the constructor parameter:

```
@Component
class FileSystem {
 FileSystem( @Value( "10" ) long minimumFreeDiskSpace ) { … }
}
```

The parameter variable of the constructor should be assigned the value `10`. Here, `@Autowired` is no longer necessary because constructors—and we've already seen this—are called automatically even without `@Autowired`. Just as with `boolean`, here we have a type conversion via the `ConversionService`: the string `"10"` becomes a `long` 10.

With `@Value`, you can only specify a string because the annotation attribute `value` is of type `String`. The first example to `@Value` looked like this:

```
@Value( "FillmoreFat" ) String nickname;
```

Although the value is a string constant, it doesn't have to be a compile-time constant. It's also possible to determine the assignment of the `nickname` variable dynamically.

3.2.4 Get Environment Assignments via @Value and ${…}

Getting the `Environment` injected and fetching the values from the environment isn't a good example of *inversion of control* (IoC). We should only ever have things given to us, not fetch them ourselves. Just as we normally never request a Spring-managed bean from the container with `getBean(…)`, we shouldn't normally fetch anything from the `Environment` with `getProperty(…)`. The right idea is to have the assignment of a property injected into a variable.

This is where `@Value` comes into play. The notation for a property access via `@Value` is `${PropertyVariable}`. There's another small variation for default values, as we'll see in a moment. Following are three examples:

```
@Value( "${user.home}" )
String userHome;

@Value( "${com.tutego.homepage}" )
String homepage;

@Value( "${com.tutego.number-of-seminars}" )
int numberOfSeminars;
```

The advantage of the notation is that the values no longer have to be actively fetched from the `Environment`. With injection, it's another form of IoC. The visibility doesn't matter because the injection is independent of the visibility just as with `@Autowired`.

Besides the syntax with the dollar sign and the property variable, there is a similar notation with a number sign instead of the dollar sign: `#{SpEL}`. This is an example of the Spring Expression Language (SpEL). As you've already seen in Chapter 2, Section 2.11, this is much more powerful because with the dollar sign, only property variables can be accessed, but with the SpEL complex expressions, multiple variables can be evaluated.

Note: Misspelling @Value

What effect would the following annotation have on the instance variable?

```
@Value( "${com.tutego.hompage}" )
private String homepage;
```

The identifier `hompage` has a small spelling mistake, and that leads to an `IllegalArgumentException`. This is like an `@Autowired` that can't be resolved, which also results in an exception.

Certain values are allowed to be unassigned and optional. With Spring, variables can be assigned a default value if the configuration properties haven't been set.

3.2.5 @Value and Default Values

A colon can be placed after the key, which can be followed by a default value. Here are a few examples:

```
@Value( "${key:my default value}" )
String stringWithDefaultValue;

@Value( "${key:}" )
String stringWithBlankDefault;

@Value( "${key:true}" )
boolean booleanWithDefault;

@Value( "${key:one,two,three}" )
String[] stringArrayWithDefaults;

@Value( "${key:1,2,3}" )
int[] intArrayWithDefaults;

@Value( "${key:#{null}}" )
String stringOrDefaultNull;
```

Behind the colon, there doesn't have to be anything, then the default value is an empty string. Conversions, for example, from a string to `boolean`, are done automatically by Spring. The `ConversionService` also allows complex conversions, for example, from comma-separated integers in the string to an `int` array.

If the reference is to become `null`, it's possible to use SpEL via a small detour.

@Value at Instance Variables or Constructor Parameters

We've already discussed the problem that a constructor must not use `@Autowired` variables that are only set later via field or setter injection. We have the same problem with `@Value`:

```
@Service
class HomepageGenerator {
  @Value( "${com.tutego.homepage}" )
  private String homepage;

  HomepageGenerator() {
    homepage = homepage.toLowerCase(); ☠
  }
}
```

The constructor is supposed to read the instance variable and convert the string to lowercase. But this doesn't work and will lead to a `NullPointerException` because the variable `homepage` is still `null` at that time. Using an `@Value` constructor injection for environment values solves the problem:

```
HomepageGenerator( @Value("${com.tutego.homepage}") String homepage ) {
  this.homepage = homepage.toLowerCase();
}
```

The framework passes the assigned value of `com.tutego.homepage` into the constructor. In addition, in the `@Bean` methods, parameters can be annotated with `@Value`.

Inject minimumFreeDiskSpace via @Value

As a reminder, here is the `FsCommands` class method.

```
@ShellMethod( "Display required free disk space" )
public long minimumFreeDiskSpace() {
  return 1_000_000;
}
```

Listing 3.2 FsCommands Excerpt

When the `minimum-free-disk-space` command is called, `1000000` is returned.

It's easy to get the value from a configuration property. First, in *application.properties* (or other property sources, as you'll see later), we set the configuration property `date4u.filesystem.minimumFreeDiskSpace` to `1000000`.

```
date4u.filesystem.minimum-free-disk-space=1000000
```

Listing 3.3 application.properties

Next, we need to modify `FsCommands`.

```
@ShellComponent
public class FsCommands {

  @Value( "${date4u.filesystem.minimum-free-disk-space:1000000}" )
  private long minimumFreeDiskSpace;
  @ShellMethod( "Display required free disk space" )
  public long minimumFreeDiskSpace() {
    return minimumFreeDiskSpace;
  }
  // …
}
```

Listing 3.4 FsCommands Extension

Note

It's a common mistake to forget the `${…}` symbols around the property name, so that it says, for example:

```
@Value( "date4u.filesystem.minimum-free-disk-space:1000000" )
private long minimumFreeDiskSpace;
```

In this case, Spring would try a `Long.parseLong("date4u.filesystem.minimum-free-disk-space:1000000")`, which results in an exception.

3.2.6 Access to Configurations

The access to the configuration properties is possible via three variants (we've already covered the first two):

- Associative memory of the `Environment` type, which contains all configurations
- Variables annotated with `@Value`, which get an assignment from Spring
- Structured object that populates Spring Boot, similar to Java Architecture for XML Binding (JAXB) or Jakarta Persistence

Interestingly, the third variant is only possible in Spring Boot, but not in pure Spring Framework applications.

Configurations often have a hierarchical structure that can be easily mapped to objects. Spring Boot extends the principle and can map `Environment` properties from all property sources to objects.

Map Configuration Data to Beans with @ConfigurationProperties

To make the mapping possible, first a classic JavaBean with setters and getters is created. Next, the class is annotated with `@ConfigurationProperties`. Here's an example:

```
@Component
@ConfigurationProperties( "com.tutego" ) // prefix of the configuration
public class TutegoConfigurationProperties {
  private String homepage;
  private int    numberOfSeminars;
  public String getHomepage() { return homepage; }
  public void setHomepage( String homepage ) {
    this.homepage = homepage;
  }
  public int getNumberOfSeminars() { return numberOfSeminars; }
  public void setNumberOfSeminars( int numberOfSeminars ) {
    this.numberOfSeminars = numberOfSeminars;
  }
}
```

For Spring Boot to know which `Environment` variables to map to the component, a prefix must be specified; all properties after the `com.tutego` prefix are mapped to the object. Anything before the prefix isn't your own configuration data and shouldn't be included.

As usual, the JavaBean has a parameterless constructor and the setters/getters. The setters/getters may be annoying, but they are central to the mapping:

1. In the first step, Spring Boot calls the constructor and builds an instance.
2. Then, Spring Boot calls the setters for all matching properties, so that the bean is filled with the `Environment` variables. Suppose the property `com.tutego.homepage` has been initialized to `http://tutego.com/`, for example, in the *application.properties* file. Then, Spring would call `setHomepage("http://tutego.com/")` on the created object.

The end result is a normal Spring-managed bean that can be wired. Here's an example:

```
@Autowired
TutegoConfigurationProperties props;
```

Then the getters can be called on the `props` object to get the data:

```
log.info( "Homepage: {}, Number of seminars {}",
          props.getHomepage(), props.getNumberOfSeminars() );
```

@ConfigurationProperties on Records

Transferring to objects is a handy thing, and it can be shortened even further with records. Although records aren't JavaBeans, records can be used in many places in Spring. With records, the boring setters/getters can be omitted, and the code becomes nicely compact. This is how `TutegoConfigurationProperties` looks with a record:

```
@Component
@ConfigurationProperties( "com.tutego" )
public record TutegoConfigurationProperties(
  String homepage,
  int numberOfSeminars
) { }
```

The record components must be determined, and the Java compiler creates a parameterized constructor from it, which Spring Boot calls later. Multiple constructors are possible for records in principle, and then the constructor that Spring Boot should call is annotated with `@ConstructorBinding`. Often, however, this doesn't happen.

Records became a standard feature in Java 16, so, of course, they can be used for Spring Boot 3, which requires Java 17. Before records, *Lombok* was popular, and that is still an option if mutable Java objects are needed because records are known to be immutable.

@DefaultValue

Because a record has no setters, a small problem arises: With a parameterized constructor, all values must be specified; with a regular JavaBean as a container for the configuration properties, it's fine if certain setters aren't called. With parameterized constructors or with records, however, all parameters must be filled.

The annotation `@DefaultValue`[5] solves the problem. Let's assume that the homepage may be missing from the record `TutegoConfigurationProperties`, and, in this case, a default value is to be set. In Java, it's not possible to set a default value for a parameter, which is why annotation `@DefaultValue` is used. Here's an example:

```
@ConfigurationProperties( "com.tutego" )
public record TutegoConfigurationProperties(
  @DefaultValue( "http://tutego.com" ) String homepage,
  int numberOfSeminars
) { }
```

Annotation `@DefaultValue` specifies the default value, so the assignment of `homepage` can be missing.

[+]

Tip

The `@DefaultValue` annotation is also useful in another place. Configuration objects can have referenced child objects, as we'll see in Section 3.2.7. If this subobject is missing, then the reference remains `null`. It may be that these subobjects need to be specified with configurations. This is where `@DefaultValue` comes in handy because it enforces that subobjects must be specified and the instance variable must not be `null`. The `@DefaultValue` annotation is used in this case without an annotation attribute.

@ConfigurationProperties at @Bean

If `@ConfigurationProperties` is applied to the `@Component` class, a Spring-managed bean is built, and, at the end, just before the component is put into the container, the setters are called, and the configuration-properties are set. With the records it looks a little differently: the parameterized constructors are called directly, and the values are filled.

---

[5] *https://docs.spring.io/spring-boot/docs/current/api/org/springframework/boot/context/properties/bind/DefaultValue.html*

The `@ConfigurationProperties` annotation can also be used at an `@Bean` method. With an `@Bean` method, we take care of building the component itself, but Spring Boot can take care of transferring the configuration properties. Here's an example where again `TutegoConfigurationProperties`—as JavaBean, not as record—is used:

```
@Configuration( proxyBeanMethods = false )
class TutegoConfiguration {
  @Bean
  @ConfigurationProperties( "com.tutego" )
  public TutegoConfigurationProperties tutegoConfigurationProperties() {
    return new TutegoConfigurationProperties();
  }
}
```

The `@Bean` method creates the `TutegoConfigurationProperties` object in the body via the parameterless constructor and doesn't set any data. But because the `@Bean` method is annotated with `ConfigurationProperties`, Spring will call the setters on this freshly built component in the second step and set the `Environment` values. For this, of course, the component must be mutable and provide setters.

The technique is handy when JavaBean classes aren't ours. We've already talked about this with `@Bean` methods: the advantage is that this way any components can get into the Spring context that aren't annotated with `@Component`. Here the principle is the same: especially for external components that can accept configurations with setters, this combination of `@Bean` and `@ConfigurationProperties` can be used well.

@EnableConfigurationProperties

The `@ConfigurationProperties` annotation has no direct relationship to the `@Component` or `@Bean` annotations. You can think of `@ConfigurationProperties` as an extended initialization.

There is a shorter way of writing, where only annotation `@ConfigurationProperties` is needed and `@Component` can be omitted. In this case, annotation `@EnableConfigurationProperties` is used. Here's an example:

```
// @Component  -- not necessary anymore
@ConfigurationProperties( "com.tutego" )
public record TutegoConfigurationProperties(
  String homepage, int numberOfSeminars
) { }
```

For `TutegoConfigurationProperties`, the compact record notation is used. The `@Component` annotation can be omitted if `@EnableConfigurationProperties` is set on an `@Configuration` class. This is what it can look like:

```
@Configuration( proxyBeanMethods = false )
@EnableConfigurationProperties({TutegoConfigurationProperties.class})
class PropertiesConfigurations { }
```

An array of `Class` objects is specified on `@EnableConfigurationProperties`, and each `Class` object represents the configuration objects. If there is only a single element in the array, the curly braces of an array can be removed as usual. Because an `@SpringBootApplication` is also an `@Configuration`, `@EnableConfigurationProperties` could easily be at a main class.

@ConfigurationPropertiesScan

`@EnableConfigurationProperties` is a good shortcut. However, if a program declares many `@ConfigurationProperties`, another shortcut can be used that searches for all types annotated with `@ConfigurationProperties`; these classes become the Spring-managed bean by themselves. This is similar to `@ComponentScan` as it declaratively expresses that special annotated classes should be found. Here's a small example:

```
package com.tutego.date4u;
…
@SpringBootApplication
@ConfigurationPropertiesScan
class Date4uApplication { … }
```

The `@ConfigurationPropertiesScan` is all about finding `@ConfigurationProperties` classes. `Date4uApplication` is the main configuration, and `@ConfigurationPropertiesScan` declares that the search is starting from the `com.tutego.date4u` package and recursively collecting everything annotated with `@ConfigurationProperties` arbitrarily deep. It also registers the class as a Spring-managed bean.

Using `@ConfigurationPropertiesScan` saves the individual enumeration that is necessary with `@EnableConfigurationProperties` and lets the scanner do the rest. Comparable to `@ComponentScan`, which we saw in Chapter 2, Section 2.3.4, `@ConfigurationPropertiesScan` lets you specify packages to search from. The package names can be specified as `String[]` or `Class<?>[]`.

Of all the notations, this variant is the shortest, but we would need active classpath scanning.

Best Practice

Around `@ConfigurationProperties` classes arise some recommended procedures. If there is a configuration property but no setter in the `@ConfigurationProperties` class, this isn't an error. Because we can access all configuration properties via the `Environment` or `@Value` anyway; the values didn't disappear. However, `ConfigurationProperties` has two `boolean` annotation attributes, `ignoreInvalidFields` and `ignoreUnknownFields` (both are `true` by default), which escalate to an error if the type or identifier is incorrect.

If validation of the values is necessary, this can be checked by hand or, even better, deferred to the Jakarta Bean Validation, because this is declarative. We'll return to validation in Chapter 4, Section 4.5. If you want to know whether values have been set at all, you can set a flag in setters.

If a setter isn't called, the property should be given a default value so that a getter later returns a meaningful value. This is easily done by initializing the instance variable:

```
@Component
@ConfigurationProperties( "com.tutego" )
public class TutegoConfigurationProperties {
  private String homepage = "http://tutego.com/";
  …
}
```

If the setter is called, the default value is overwritten; if the setter isn't called, this default value is returned.

The combination of `@ConfigurationProperties` objects and `@Component` makes the objects full-fledged Spring-managed beans, but that's not how they should be used. That is, the components should just express values of an `Environment`, but not become complex business objects in which you wire services or the like. In other words, on one hand, you have `@ConfigurationProperties` as a container exclusively for the data, and on the other hand, you have the services that use these `@ConfigurationProperties` objects.

3.2.7 Nested/Hierarchical Properties

Properties are often nested; that is, after one hierarchy, there is another hierarchy, followed by another, and so on. Let's look at two examples of some selected Spring Boot properties, one in properties format and one in YAML format:

- Properties format:

  ```
  spring.main.banner-mode
  spring.mail.username
  spring.rsocket.server.port
  spring.data.web.pageable.default-page-size
  ```

- YAML format:

  ```
  spring:
    mail:
      username: …
  ```

```
    main:
      banner-mode: …
    data:
      web:
        pageable:
          default-page-size: …
    rsocket:
      server:
        port: …
```

The YAML format shows very nicely the tree-like structure, which you can't capture so well with the properties file.

The meaning of the configurations isn't important here, it's just important to realize that configuration properties aren't necessarily flat

Note

By the way, all Spring properties are listed with explanation at *https://docs.spring.io/spring-boot/docs/current/reference/html/application-properties.html*.

Implement Nested Properties

Nested properties are achieved through referenced child objects:

```
@Component
@ConfigurationProperties( "com" )
public class TutegoConfigurationProperties {
  private Tutego tutego;
  public Tutego getTutego() { return tutego; }
  public void setTutego( Tutego tutego ) { this.tutego = tutego; }
}

class Tutego {
  private String homepage;
  private int numberOfSeminars;
  public String getHomepage() { return homepage; }
  public void setHomepage( String homepage ) {
    this.homepage = homepage;
  }
  public int getNumberOfSeminars() { return numberOfSeminars; }
  public void setNumberOfSeminars( int numberOfSeminars ) {
    this.numberOfSeminars = numberOfSeminars;
  }
}
```

What the code implements is the following: there are still properties under the `com.tutego` branch, but it's no longer `@ConfigurationProperties( "com.tutego" )`; instead, it's `@ConfigurationProperties( "com" )`, and `TutegoConfigurationProperties` refers to an object of type `Tutego`. The main class has the usual setters and getters. The class `Tutego` takes over the members below `com.tutego`, namely `homepage` and `numberOfSeminars`. Further members could lie beyond that.

Task: Introduction of a Configuration Class

In Section 3.2.5, we wrote the following in the `FsCommands`:

```
@Value( "${date4u.filesystem.minimum-free-disk-space:1000000}" )
private long minimumFreeDiskSpace;
```

You're welcome to replace `@Value` with a free space configuration class as an exercise.

Proposed solution: First, create a new configuration class; let's call it `Date4uProperties`. The prefix is `date4u`. Class `Date4uProperties` contains nested class `Filesystem`. We don't want to call the class `FileSystem` because the camel case class name becomes the property name `file-system`, but `filesystem` is nicer to write.

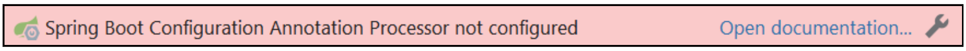
```
@Component
@ConfigurationProperties( "date4u" )
public class Date4uProperties {

  public static class Filesystem {
    private long minimumFreeDiskSpace = 1_000_000;
    public long getMinimumFreeDiskSpace() {
      return minimumFreeDiskSpace;
    }
    public void setMinimumFreeDiskSpace( long minimumFreeDiskSpace ) {
      this.minimumFreeDiskSpace = minimumFreeDiskSpace;
    }
  }

  private Filesystem filesystem = new Filesystem();
  public Filesystem getFilesystem() {
    return filesystem;
  }
  public void setFilesystem( Filesystem filesystem ) {
    this.filesystem = filesystem;
  }
}
```

The initialization with `filesystem = new Filesystem()` is important because if you inject a `Date4uProperties` object and call `getTutego().getMinimumFreeDiskSpace()` on it, there should be no `NullPointerException`.

The setters/getters can be a bit confusing with the nested classes. At this point, you can understand the purpose of Lombok, which hides all the setters and getters from us.

Configuration Metadata Annotation Processor

If a `@Configuration` class is open in IntelliJ, there is a message at the beginning above the Java editor, as shown in Figure 3.2.



Spring Boot Configuration Annotation Processor not configured

Figure 3.2 IntelliJ Indicating the Installation of an Annotation Processor

An *annotation processor* is a kind of compiler plugin. The special *Spring Boot Configuration Annotation Processor* writes metadata to a JavaScript Object Notation (JSON) file. The file documents the following for each property:

- Name (e.g., `minimumFreeDiskSpace`)
- Data type (in our case, `long`)
- Short description (from the Javadoc "required minimum free disk space for local file system")
- Other validation information from Jakarta Validation (which we don't use).

The special annotation processor isn't configured by default and is activated, for example, by an entry in the project object model (POM) file. Clicking **Open Documentation** in IntelliJ brings the browser to *https://docs.spring.io/spring-boot/docs/current/reference/html/configuration-metadata.html#appendix.configuration-metadata.annotation-processor* and shows the snippet that needs to be in the POM file. However, that is just one option. It is better to declare the annotation processor in the Maven compiler configuration, as this allows multiple annotation processors to be placed in their correct order:

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <configuration>
        <annotationProcessorPaths>
          <path>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-configuration-processor</artifactId>
            <version>${project.parent.version}</version>
          </path>
        </annotationProcessorPaths>
      </configuration>
    </plugin>
    <!-- spring-boot-maven-plugin -->
  </plugins>
</build>
```

Listing 3.5 pom.xml Extension

After insertion in the POM file, the Maven compile phase can be triggered, and a newly generated file appears in the *target directory*:

target\classes\META-INF\spring-configuration-metadata.json

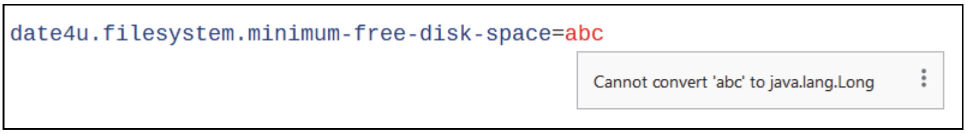
In the file, the property can be found again, including the data type, the section from the Javadoc, and the default value:

```
{
  "groups": [
    {
      "name": "date4u",
      "type": "com.tutego.date4u.core.configuration.Date4uProperties",
      "sourceType": "com.tutego.date4u.core.configuration.Date4uProperties"
    },
    {
      "name": "date4u.filesystem",
      "type": "com.tutego.date4u.core.configuration
.Date4uProperties$Filesystem",
      "sourceType": "com.tutego.date4u.core.configuration.Date4uProperties",
      "sourceMethod": "getFilesystem()"
    }
  ],
  "properties": [
    {
      "name": "date4u.filesystem.minimum-free-disk-space",
      "type": "java.lang.Long",
      "description": "Required minimum free disk space for local file system.",
      "sourceType":
"com.tutego.date4u.core.configuration.Date4uProperties$Filesystem",
      "defaultValue": 1000000
    }
  ],
  "hints": []
}
```

The file format is described on the Spring website:

https://docs.spring.io/spring-boot/docs/current/reference/html/configuration-metadata.html

A development environment can access this information even if the source code of the `@ConfigurationProperties` class with the documentation isn't available at all. The settings also help with keyboard completion, and development environments can perform type checking. For example, if our integer property is invalidly assigned, there will be an error. So, it's handy to have the development environment do the type checking because that reduces problems later on, as Figure 3.3 shows.



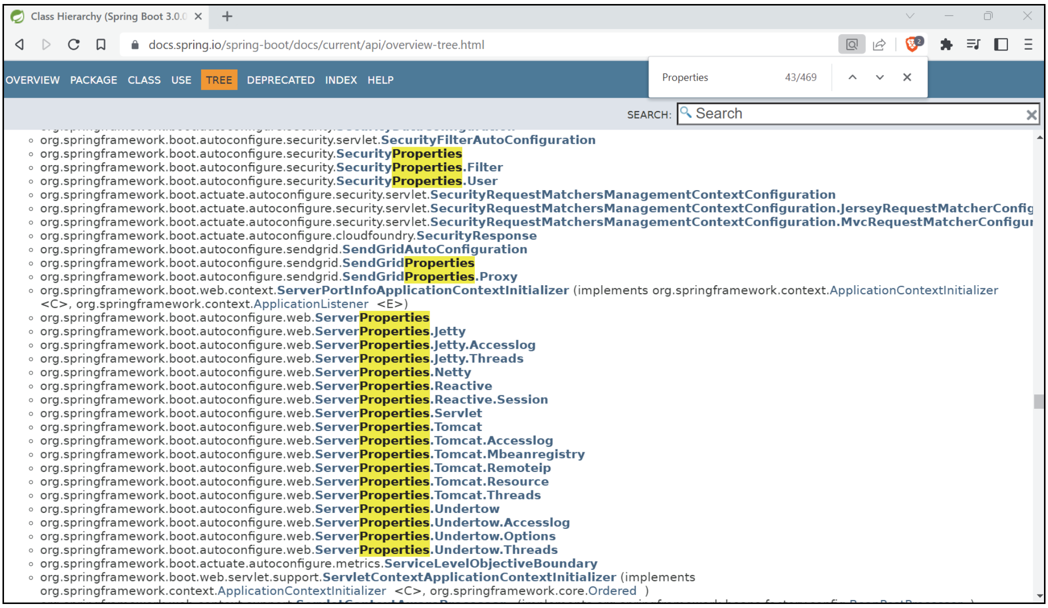
```
date4u.filesystem.minimum-free-disk-space=abc
```

Cannot convert 'abc' to java.lang.Long

Figure 3.3 IntelliJ Checking the Types

@ConfigurationProperties from Spring Boot

Spring Boot relies exclusively on `@ConfigurationProperties` classes, and the Javadoc[6] shows this vividly: a search for "properties" yields countless results (see Figure 3.4).


Class Hierarchy (Spring Boot 3.0.
docs.spring.io/spring-boot/docs/current/api/overview-tree.html
OVERVIEW
PACKAGE
CLASS
USE
TREE
DEPRECATED
INDEX
HELP
Properties
43/469
SEARCH:
Search

- org.springframework.boot.autoconfigure.security.servlet.**SecurityFilterAutoConfiguration**
- org.springframework.boot.autoconfigure.security.**SecurityProperties**
- org.springframework.boot.autoconfigure.security.**SecurityProperties.Filter**
- org.springframework.boot.autoconfigure.security.**SecurityProperties.User**
- org.springframework.boot.actuate.autoconfigure.security.servlet.**SecurityRequestMatchersManagementContextConfiguration**
- org.springframework.boot.actuate.autoconfigure.security.servlet.**SecurityRequestMatchersManagementContextConfiguration.JerseyRequestMatcherConfig**
- org.springframework.boot.actuate.autoconfigure.security.servlet.**SecurityRequestMatchersManagementContextConfiguration.MvcRequestMatcherConfigu**
- org.springframework.boot.actuate.autoconfigure.cloudfoundry.**SecurityResponse**
- org.springframework.boot.autoconfigure.sendgrid.**SendGridAutoConfiguration**
- org.springframework.boot.autoconfigure.sendgrid.**SendGridProperties**
- org.springframework.boot.autoconfigure.sendgrid.**SendGridProperties.Proxy**
- org.springframework.boot.web.context.**ServerPortInfoApplicationContextInitializer** (implements org.springframework.context.ApplicationContextInitializer <C>, org.springframework.context.ApplicationListener <E>)
- org.springframework.boot.autoconfigure.web.**ServerProperties**
- org.springframework.boot.autoconfigure.web.**ServerProperties.Jetty**
- org.springframework.boot.autoconfigure.web.**ServerProperties.Jetty.Accesslog**
- org.springframework.boot.autoconfigure.web.**ServerProperties.Jetty.Threads**
- org.springframework.boot.autoconfigure.web.**ServerProperties.Netty**
- org.springframework.boot.autoconfigure.web.**ServerProperties.Reactive**
- org.springframework.boot.autoconfigure.web.**ServerProperties.Reactive.Session**
- org.springframework.boot.autoconfigure.web.**ServerProperties.Servlet**
- org.springframework.boot.autoconfigure.web.**ServerProperties.Tomcat**
- org.springframework.boot.autoconfigure.web.**ServerProperties.Tomcat.Accesslog**
- org.springframework.boot.autoconfigure.web.**ServerProperties.Tomcat.Mbeanregistry**
- org.springframework.boot.autoconfigure.web.**ServerProperties.Tomcat.Remoteip**
- org.springframework.boot.autoconfigure.web.**ServerProperties.Tomcat.Resource**
- org.springframework.boot.autoconfigure.web.**ServerProperties.Tomcat.Threads**
- org.springframework.boot.autoconfigure.web.**ServerProperties.Undertow**
- org.springframework.boot.autoconfigure.web.**ServerProperties.Undertow.Accesslog**
- org.springframework.boot.autoconfigure.web.**ServerProperties.Undertow.Options**
- org.springframework.boot.autoconfigure.web.**ServerProperties.Undertow.Threads**
- org.springframework.boot.actuate.autoconfigure.metrics.**ServiceLevelObjectiveBoundary**
- org.springframework.boot.web.servlet.support.**ServletContextApplicationContextInitializer** (implements org.springframework.context.ApplicationContextInitializer <C>, org.springframework.core.Ordered)

Figure 3.4 Javadoc Output: Searching for "Properties"

---

[6] *https://docs.spring.io/spring-boot/docs/current/api/*

Knowing this can be useful because not every configuration property is explained in detail; then looking at the `@ConfigurationProperties` implementation can help. The Javadoc documents the nested classes that are used for subobjects.

3.2.8 Map Special Data Types

The mappings used so far have been relatively simple. The next pages show that more complex data types are also possible.

Arrays and Lists

Arrays and lists can be configured in the *application.properties* file. For instance, consider a list of IP addresses stored as strings in `date4u.servers`. The corresponding configuration in properties format would be the following:

```
date4u.servers[0]=126.32.66.28
date4u.servers[1]=111.229.178.175
date4u.servers[2]=4.16.177.175
```

For a list, you use square brackets with an index that specifies where this entry is placed. That means, `date4u.servers[0]` stands for the first element.

In YAML format, the entries of a sequence are preceded by a minus sign:

```
date4u:
    servers:
        - 126.32.66.28
        - 111.229.178.175
        - 4.16.177.175
```

In a subsection of Section 3.2.10, YAML files are introduced once again.

If we wanted to capture that on the Java side, we would write the following:

```
@ConfigurationProperties( "date4u" )
public class Date4uProperties {
  private List<String> servers = new ArrayList<>();
  // Setter + Getter
}
```

The `value` annotation attribute of `@ConfigurationProperties` is assigned with `date4u`, and `servers` is then of type `List`. Before, we've always used simple data types such as `String` or `long`, and this time we use a data structure. Of course, setters/getters must still exist, but they are hidden here. The mapping also works with a record, of course.

Referenced Subobjects in Lists

The configuration list can also reference subobjects instead of just strings, numbers, and so on. To demonstrate this, let's modify the configuration file by adding an `ip` property:

```
date4u.servers[0].ip=126.32.66.28
date4u.servers[1].ip=111.229.178.175
```

Alternatively, in YAML format, it looks like this:

```
date4u:
    servers:
        -  ip: 126.32.66.28
        -  ip: 111.229.178.175
```

YAML shows the difference more clearly that an additional object is added which contains the IP address.

Here, the same is shown in Java:

```
@Component
@ConfigurationProperties( "date4u" )
public class Date4uProperties {
  record Server( String ip ) { }
  private List<Server> servers = new ArrayList<>();
  // Setter + Getter for servers
}
```

The difference with the solution before is that a new container is needed for the server's information; the program uses a compact record, and a class with setters and getters would also be valid.

The list doesn't reference strings as before, but `Server` objects. Setters and getters for `servers` are necessary again. At the end, there is a list of `Server` objects, each storing an IP address.

Save Properties by Key

A list is basically a special associative memory where the keys are numeric values greater than or equal to 0. In addition to lists, Spring can also map data into `Map` objects. It doesn't have to be a mapping from a numeric index to an element; a mapping from a string (key) to an object (value) is also valid.

The preceding example, rewritten a bit more, looks like this—first in properties format:

```
date4u.servers.home-server.ip=126.32.66.28
date4u.servers.traffic-server.ip=125.33.43.2
```

Here, it is in YAML format:

```
date4u:
  servers:
    home-server:
      ip: 126.32.66.28
    traffic-server:
      ip: 125.33.43.2
```

Previously, the file used the indices 0, 1, and 2, but now it uses the server names `home-server` and `traffic-server`. Afterwards, because they should be `Server` objects again, the actual assignment follows with the IP address.

In Java terms, it's only a small difference from before because `List` becomes `Map`:

```
@Component
@ConfigurationProperties( "date4u" )
public class Date4uProperties {
  record Server( String ip ) { }
  private Map<String, Server> servers = new HashMap<>();
  // Setter + Getter for servers
}
```

With a list, we have an index 0, 1, 2, and with the `Map`, the key (type `String`) is associated with a `Server` object. Of course, we could have omitted the `Server` object and directly referenced the strings with the IP address, and then it would be a `Map<String, String>`.

Setters and getters are still necessary for the instance variable `servers` of type `Map`.

Capture Arbitrary Properties

If it's not known what the key-value pairs are called, then a generic `Map<String, Object>` can be used, which associates the key (type `String`) with any object. (If it's known what type the associated objects are, then the concrete data type can be used of course.)

Let's look at an example, starting on the Java side:

```
@Component
@ConfigurationProperties( "date4u" )
public class Date4uProperties {
  private Map<String, Object> config = new HashMap<>();
  // Setter + Getter for config
}
```

For `config`, there are setters/getters, and together with the `Map<String, Object>`, any keys and associated values are possible under hierarchy `date4you.config`.

Here, we see this with a small example in the properties format:

```
date4u.config.noclue=123
date4u.config.really.no.clue=123
```

Type Conversions

The data types we've discussed so far have been strings, numeric values, lists, and maps. Spring offers the special feature that a wide variety of data types can be used because Spring internally uses converter classes. These conversions can come, for example, over *property editors* or over an internal `ConversionService` (more about this follows in Section 3.6). These services are very powerful and contain a myriad of different conversions from strings to target types.

Now, we'll examine two examples. Suppose we have a comma-separated list of numbers under the key `date4u.ports-to-test` and an assignment of a `TimeUnit` under the key `date4u.repetition-unit`:

```
date4u.ports-to-test=80,8000,8080
date4u.repetition-unit=SECONDS
```

The Java class that can capture the values may look like this:

```
@ConfigurationProperties( “date4u” )
public class Date4uProperties {
  private int[] portsToTest;
  private java.util.concurrent.TimeUnit repetitionUnit;
  // Setter + Getter
}
```

It's valid to write a comma-separated list of numbers (in our case, `80,8000,8080`). Spring will comma-separate the `String`, convert the values one by one, and convert them to an `int` array.

The second example shows that enumerations can be used. The values must have the names of the constants. `SECONDS` is an enumeration element of `TimeUnit`.

There are quite a number of these type conversions. Next, I would like to introduce three special type converters that are handy for configuration files: `DataSize`, `Duration`, and `Period`.

Converter for DataSize

Often information about sizes is needed in applications, for example, to determine a maximum buffer size or a maximum size when uploading. To represent sizes, Spring declares the `DataSize` data type.[7] The internal converters can automatically convert a string from the configuration into `DataSize` objects.

---

[7] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/util/unit/DataSize.html*

Here's an example: In the configuration, a property `data4u.history-max-file-size` should be defined with 1 MB:

```
data4u.history-max-file-size=1MB
```

Spring can convert the specification into a `DataSize` object:

```
@Value( "${data4u.history-max-file-size}" )
DataSize historyMaxFileSize;
```

To keep it short, the example omits an `@ConfigurationProperties` class, but the `DataSize` data type is easy to read.

If a string is to be converted to a `DataSize` object, then Spring allows and recognizes different suffixes in the string `B`, `KB`, `MB`, `GB`, `TB`.

It can be handy, especially for `@ConfigurationProperties` classes, to set the unit. For this purpose, `@DataSizeUnit` in combination with an enumeration element from `DataUnit` is used. Let's assume that `history-max-file-size` should default to 10 MB, and all specifications should be in the order of megabytes:

```
@ConfigurationProperties( "date4u" )
public class Date4uProperties {

  @DataSizeUnit( DataUnit.MEGABYTES )
  private DataSize historyMaxFileSize = DataSize.ofMegabytes( 10 );

  // Setter + Getter
}
```

Later, for example, in *application.properties* we have the following where `1` is assumed to be in megabytes, and `getHistoryMaxFileSize()` subsequently returns 10485760 bytes:

```
data4u.history-max-file-size=1
```

Duration Converter

Often durations are needed in applications, for example

- For the determination of timeouts
- In polling, to set the frequency at which a service should be polled
- For cache properties, for example, how long something should stay in the cache

The Java library declares the `java.time.Duration` type for durations. By default, Spring can convert a `String` into a `Duration` object. Different formats for representing a duration are possible:

- **ISO 8601 format**
  `PT0.5S` represents half a second, for example. Details about the ISO format are given in the Javadoc.[8]
- **ISO 8601**
  This format isn't well readable. Therefore, Spring allows suffixes to more easily specify nanoseconds, milliseconds, seconds, minutes, hours, and days. The suffixes are `ns`, `ms`, `s`, `m`, `h`, and `d`, respectively.

  If a unit isn't specified, milliseconds are automatically assumed. This means that the specifications `500`, `PT0.5S`, and `500ms` are identical.

By default, the time unit used is milliseconds, but this can be changed using the `@DurationUnit` annotation. For example, if we have a session and want to set the session timeout in seconds, we can do the following:

```
@ConfigurationProperties( "date4u" )
public class Date4uProperties {

  @DurationUnit( ChronoUnit.SECONDS )
  private Duration sessionTimeout = Duration.ofSeconds( 20 );

  // Setter + Getter
}
```

The getter returns the default value of 20 seconds if the setter isn't called. If a session timeout is set, then this is done in the seconds unit.

Converter for Period

Another data type for which Spring provides a converter is `java.time.Period`. `Period` and `Duration` are related; the difference is that `Duration` stores intervals between times, and `Period` stores intervals between dates.

A Java `Period` object can be specified in the ISO 8601 notation. There is also the possibility to work with `y`, `m`, `w`, and `d`: `2y2d` then stands for the interval of 2 years and 2 days. Just as a unit can be specified for `@DurationUnit` and `@DataSizeUnit`, the unit can be specified for `Period` and `@PeriodUnit`; the default is days.

3.2.9 Relaxed Bindings

In Java, it's common to use camel case notation for identifiers, except for constants. On the other hand, configuration files often use hyphenated notation between members. However, what is written with a hyphen in a property isn't allowed for a Java variable name.

---

[8] *https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/time/Duration.html#parse(java.lang.CharSequence)*

When converting, Spring does more than just adjust the types via the converters. Certain differences between the property name and the Java code are allowed; Spring calls this *relaxed binding*.

Property	Designation
`com.tutego.numberOfSeminars`	Standard camel case syntax
`com.tutego.number-of-seminars`	Kebab case; recommended for *.properties* and *.yml* files
`com.tutego.number_of_seminars`	Snake case (underscore notation); an alternative for the files
`COM_TUTEGO_NUMBEROFSEMINARS`	Uppercase format; recommended for environment variables

Table 3.1 Different Property Notations Allowed with Relaxed Binding

Spring automatically maps the examples on the left side of Table 3.1 to the corresponding setters, and in no case, does the method name need to be adjusted. A bit more unusual is the notation for environment variables, which are traditionally capitalized. If everything is capitalized, the dot is replaced by an underscore.

@Value versus @ConfigurationProperties

You've learned a total of three variants for reading the `Environment` states. At this point, we want to highlight the differences between `@Value` and `@ConfigurationProperties`:

- The annotations are coming from different packages: the `@Value` annotation—just like the `Environment`—comes from the Spring Framework. `@ConfigurationProperties`, on the other hand, is a feature of Spring Boot.
- In relaxed binding, case doesn't really matter for properties, and mixed case isn't necessary for mapping to setters in camel case. However, if something is injected with the `@Value` annotation, the case must be exact. That is, if a property contains minus signs, the string must be exactly the same at `@Value`.
- With `@ConfigurationProperties` applied, an annotation processor can write a JSON file for metadata support, and this in turn is evaluated by the development environment. Information such as data type, description, and validation rules come from the JavaBean. If a value is injected with `@Value`, then the whole type description doesn't exist at all because there is no bean.
- `@Value` annotations allow `#{…}` expressions in SpEL in addition to `${…}` to access the `Environment` variables. The SpEL expression is evaluated, and the evaluated result is placed in the variable.

Now that we've learned a whole bunch of interesting ways our Java application can receive the configuration information, let's look at the sources for these configurations.

3.2.10 Property Sources

A Spring Boot application obtains information from a whole set of property sources:

- **Command-line parameters via** `--`
  Configurations can be specified via the command line; for this reason, the `args` from `main(String[] args)` must be passed at `run(…)` when starting the container.
- **Inline JSON**
  This is an evaluation of an environment variable `SPRING_APPLICATION_JSON` with inline JSON.
- **Java system properties (`System.getProperties()`)**
  The properties from `System.getProperties()` are also a property source and part of `Environment`. New properties can be introduced with the `-D` command-line parameter. For example, `System.getProperties()` data includes the username.
- **Operating system environment variables (`System.getEnv()`)**
  This information is more system-specific than the Java system properties.
- **`RandomValuePropertySource` for returning random numbers**
  This property source is a bit exotic because we don't determine a fixed value, but a variable `random` helps to get, for example, a Universally Unique Identifier (UUID) or other random numbers.
- ***application.properties* file (standard property format of Java)**
  We've seen this file in the past. The key-value pairs are usually in the format `key=value` in the file.
- ***application.yml* file (YAML file format)**
  Besides property files, Spring Boot supports the alternative file format YAML.
- **Custom files enumerated using `@PropertySource` and then attached to a `@Configuration`**
  This can be useful if one *application.[properties|yml]* file isn't sufficient.

In addition, there are other profiles we'll look at later in Section 3.2.11, so that more files can be added.

This listing of property sources is almost complete, but a few exotics are missing. The complete list can be found in the reference documentation at *https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#features.external-config*.

Set Properties via the Command Line

Setting the properties from the command line is possible in two ways, and it's important to understand the difference:

- With `-Dkey=value` (no space after `-D`):

```
$ java -Dcom.tutego.homepage="http://tutego.com/" ↩
        -jar target/date4u-0.0.1-SNAPSHOT.jar
```

- With `--key=value`:

```
$ java -jar target/date4u-0.0.1-SNAPSHOT.jar ↩
        --com.tutego.homepage="http://tutego.com/"
```

The processing takes place in two completely different places, and this is evident where exactly the syntax is used:

- If a key-value pair is set in front with `-D`, the Java virtual machine (JVM) gets the value and puts it into the system properties. By the way, all Java tools can be further configured with the `-D` option, for example, also `javac` and `javah`.
- The JVM can't process options with `--`. Spring Boot itself separates the arguments that are passed on the command line. Therefore, the specification must be at the very end of the call via `java -jar`.

Reversing the two notations would be wrong and wouldn't work: If the option with `-D` is in the back, Spring Boot doesn't parse it; and if the notation is with `--` in the front, the JVM would probably not support the option and throw an error—this can change the behavior quite a bit with `--dry-run` or `--enable-preview`.[9]

Note

By default, the options passed via `--` will be part of `Environment`. If this isn't intended, for example, because you only want to evaluate the options, but the values have nothing to do in `Environment`, then the behavior can be turned off with `setAddCommandlineProperties(false)` on the `SpringApplication` object.

Spring Boot programs can also be started via Maven/Gradle. Configuration properties in this case are determined via `-Dspring-boot.run.arguments` because a regular switch with `-D` would only be forwarded to Maven/Gradle, but not necessarily to our Java program, which is usually started in a separate process by Maven/Gradle.

Here's an example:

```
$ mvn spring-boot:run ↩
      -Dspring-boot.run.arguments="--arg1=value1 --arg2=value2"
```

More about the `spring-boot` plugin is provided in Chapter 11, Section 11.1.4.

SPRING_APPLICATION_JSON

The next possible configuration source is the content of environment variable `SPRING_APPLICATION_JSON` or the assignment of configuration property `spring.application.json` with a JSON string as a one-liner. Spring Boot automatically evaluates the JSON string and translates it into properties, which then appear in `Environment`.

---

[9] *https://docs.oracle.com/en/java/javase/17/docs/specs/man/java.html*

Here's an example:

```
SPRING_APPLICATION_JSON='{"com":{"tutego":{"homepage": "http://tutego.com/"}}'
```

This JSON string contains an assignment for `com.tutego.homepage` to the homepage. The advantage is that a single environment variable can set a large number of properties in one go.

If the environment variable isn't welcome, the configuration property provides a second way to assign it. So, it could be set in principle:

```
$ java -Dspring.application.json='{…}' -jar ….jar
$ java -jar app.jar --spring.application.json='{…}'
```

Random Numbers

A `RandomValuePropertySource` under the name `random` returns random numbers. Here's an example from the reference documentation:[10]

```
my.secret=${random.value}
my.number=${random.int}
my.bignumber=${random.long}
my.uuid=${random.uuid}
my.number.less.than.ten=${random.int(10)}
my.number.in.range=${random.int[1024,65536]}
```

YAML Files

We've always used properties files for our configuration in the past. However, as discussed earlier, Spring Boot also supports YAML (*https://yaml.org*) as a second file format. YAML is a human-readable data serialization format that allows for easy representation of hierarchical structures similar to JSON or XML.

One of the main advantages of YAML over other file formats is its simplicity and lack of unnecessary syntax. Unlike XML or JSON, YAML doesn't require the use of curly or angle brackets to define the structure of the data. This makes YAML files more concise and easier to read and write.

YAML files consist of key-value pairs, which can be nested to form complex data structures. The format also supports lists and maps, making it a versatile tool for storing and manipulating data. In summary, YAML files offer a streamlined and flexible way of configuring applications, and we've already seen some examples of its use in Spring Boot.

Consider the following two configurations for turning off the banner and setting the application name:

---

[10] *https://docs.spring.io/spring-boot/docs/current/reference/html/features.html#features.external-config.random-values*

- **Possibility 1**: In *application.properties*:

```
spring.main.banner-mode=off
spring.application.name=Best application in the world.
```

- **Possibility 2**: In *application.yml*:

```
spring:
    main:
        banner-mode: "off"
    application:
        name: Best application in the world.
```

In properties files, the dot is used as a hierarchy separator. That is, there is `spring` as the root object and below it the `main` or `application` node, and below that are the actual properties. In YAML files, a colon is placed after a hierarchy, followed by an indentation.

Spring considers YAML files and properties files to be equivalent, although strictly speaking, a properties file has a slightly higher priority than the YAML file. You have the option to write in either file, and there are fans of both file formats.

Warning

The YAML format has some peculiarities that you need to know. A good example is

```
banner mode: off
```

The string `off` is interpreted as a truth value, and, in the end, the YAML parser returns `banner-mode` = `false`. This means that in YAML, certain terms such as “on”, “off”, or “no” have semantics and are converted to `boolean` values. If you want to work with YAML files, you have to know these peculiarities. See *https://noyaml.com/* for documentation of the problems, among which is also the Norway problem:

```
NI: Nicaragua
NL: Netherlands
NO: Norway
```

What is the error?

Search Locations for Configuration Files

So far, we’ve always put our configuration files *application.[properties|yml]* in the classpath. However, Spring Boot not only looks for these configuration files in the classpath, but there are different locations where these configuration files can be, as follows:

- 1a) *file:./config/*/* — in all direct subdirectories under */config/* where the application was started
- 1b) *file:./config/* — in the */config* subdirectory, where the application was started.
- 1c) *file:/* — in the current directory where the application was started

- 2a) *classpath:/config/* — in package */config* from classpath
- 2b) classpath:/ — in the classpath

This list is divided into 1 and 2: in 1, the files are in the file system; in 2, the files are in the classpath. At first glance, this doesn't make much difference, but keep in mind that files in the classpath are usually packaged in a Java archive (JAR), so changes to the files aren't possible. Case 2b deals with our file in target directory *target/classes/application.properties*.

Multiple configuration files in different locations are allowed. So, it's possible to have a configuration file in the *config* directory and in the classpath as well. Both files are loaded and taken into account. The order in the preceding list indicates the priority: entries at the top have a higher priority and consequently cover everything from the other configurations.

If there is an *application.properties* and an *application.yml* file at the same time, the values from the properties file overwrite the values from the YAML file. It's important that both documents are read and processed anyway.

Note: Question of Understanding

What is the assignment of `com.tutego.homepage` if the following files contain the specified content?

- ***file:./config/application.properties***

  `com.tutego.homepage=https://tut.com/`
- ***file:./config/http/application.properties***

  `com.tutego.homepage=http://tute.com/`
- ***file:./config/http/s/application.properties***

  `com.tutego.homepage=https://tuteg.com/`
- ***classpath:application.properties***

  `com.tutego.homepage=https://tutego.com/`

Resolution: The first file isn't considered at all because it's only searched in direct subdirectories of the *config* directory. The same applies to *config/http/s/application.properties* because this is too “far away” and is also not read. A look at the list shows that *file:/config/*/* has the highest priority, which, in our case, is *config/http/application.properties*. Consequently, `com.tutego.homepage` is equal to *http://tute.com/*. What is normally in *application.properties* has the weakest priority of all. That is, the entry isn't considered at all, and everything else is more important.

spring.config.name and spring.config.[additional-]location

We've just seen where Spring Boot looks for the configuration files. In addition, by default, Spring Boot assumes the file name *application* for the configuration files. The location can be changed just like the file name.

The configuration property `spring.config.name` changes the name. Here's an example:

```
$ java ↩
   -Dspring.config.name=demo-application ↩
   -jar app.jar
```

The file name is changed to `demo-application`.

The path, more precisely, the complete file name, can be overridden by configuration property `spring.config.location`. It's important in this context that the search location isn't extended, but the specification replaces the default search locations of Spring Boot with its own specification.

`spring.config.location` can contain a comma-separated list of files and paths, respectively. Here is a fictitious example:

```
$ java ↩
   -Dspring.config.location= ↩
    D:\date4u-cloud.properties, ↩
    classpath:/date4u.properties ↩
   -jar app.jar
```

In addition to being able to override Spring Boot's default directories with your own locations, search locations can also be added to the search locations that already exist. For this, the configuration property `spring.config.additional-location` is used; the specifications have a higher priority.

spring.config.import

It's one thing to externally change paths to configuration files via a configuration property. It's also possible to load additional configuration files at runtime by using `spring.config.import`; if this file is referenced multiple times, it will be loaded only once. This is how the specification could look:

- In *application.properties*:

  ```
  spring.config.import=datasource.properties
  ```

- In *application.yml*:

  ```
  spring:
    config:
      import: datasource.properties
  ```

Tip

`spring.config.import` can also be used to load configurations without a file extension:

```
spring.config.import=file:/etc/config/database[.yml]
```

The reference documentation call `[.yml]` an *extension hint*; it's a hint for the Spring Boot parser about the file format. The configuration file itself in our example is called *database*.

Optional Paths

We've just seen that it's possible to include more files by using various additional configuration properties. If these files aren't found, the result is a `ConfigDataLocationNotFoundException`. This behavior can be changed in two ways.

- Set `spring.config.on-not-found=ignore` so that a nonexisting file is ignored. This is a global setting.
- Prefix `optional:` can be used in front of the respective path to express that this file is optional. Optional files can be used for the following:
  - `spring.config.location`
  - `spring.config.additional-location`
  - `spring.config.import`

Here's an example:

```
$ java ↩
   -Dspring.config.location= ↩
    D:\date4u-cloud.properties, ↩
    optional:classpath:/date4u.properties, ↩
    optional:classpath:/date4u-migration.properties ↩
   -jar app.jar
```

The two files *date4u.properties* and *date4u-migration.properties* in the classpath are optional, but *D:\date4u-cloud.properties* must exist.

Config Tree

Until now, multiple configuration properties have always come from one or more files, and it's common for a file to determine more than one configuration property. However, there is an alternative way, and it's called a *config tree* in Spring. In a config tree, the key is formed from the file path, and the value of the associated key is the file content. Something like this is handy in Kubernetes, for example, because the information can be included in a file tree.

Assume the following directory structure:

- etc
  - config
    - database

  - username
  - password
- messaging
  - name

Under *etc* is the subdirectory *config*, and under that is the subdirectory *database*. The *database* folder contains the *username* and *password* files. Under *etc/config* is another folder *messaging* with a file called *name*.

If a configuration is imported with `configtree:/etc/config`, there will be three new properties:

- `database.username`
- `database.password`
- `messaging.name`

The assigned value for these properies is the file content.

Placeholder in Properties

Properties can reference other properties, which is useful for hierarchical path specifications where a root path may be defined in another variable. For instance, consider the following example:

- As a properties file:

```
com.tutego.homepage=http://tutego.com/
com.tutego.trainings=${com.tutego.homepage}/seminare
# com.tutego.trainings=${com.tutego.homepage:http://tutego.com/}/seminare
```

- As a YAML file:

```
com:
    tutego:
        homepage: http://tutego.com/
        trainings: ${com.tutego.homepage}/seminars
```

Here, key `com.tutego.homepage` is given, which is assigned a URL. A second configuration property, `com.tutego.trainings`, refers to the content of variable `com.tutego.homepage` with the dollar sign and the curly bracket. Default values can be separated with a colon, as we did, for example, with the `@Value` annotation in Section 3.2.3.

This variable substitution works automatically. The `PropertySourcesPlaceholderConfigurer` is responsible for this. If references can't be resolved, an `IllegalArgumentException` of the message `Could not resolve placeholder 'variable' in value "abc${variable}def"` follows.

Maven Resource Filtering: @…@*

Another notation, ; it's called *Maven resource filtering*, allows access to Maven or Gradle variables . A fallback to the Maven properties is possible with an `@` sign at the beginning and an `@` sign at the end of a Maven property name, provided that a POM uses `spring-boot-starter-parent`.[11]

Consider an example in *application.properties*:

```
com.tutego.source-encoding=@project.build.sourceEncoding@
com.tutego.date4u.java-version=@java.version@
com.tutego.h2.version=@h2.version@
```

In Maven, property `project.build.sourceEncoding` contains the source encoding. Using the `@` sign, we can access this variable and use it to initialize our own property—here `com.tutego.source-encoding`. Maven stores the Java version under property `java.version`; this initializes `com.tutego. date4u.java-version`. The Spring dependencies introduce a lot of new property names, such as `h2.version` for the version of the database H2; we could store that in `com.tutego.h2.version`.

The fallback can be done as usual; the version numbers may already be different:

```
@Value( "${com.tutego.source-encoding}" )
String encoding;                    // z. B. UTF-8
@Value( "${com.tutego.date4u.java-version}" )
String javaVersion;                 // z. B. 17
@Value( "${com.tutego.h2.version}" )
String h2Version;                   // z. B. 2.2.222
```

@PropertySources

We've learned so far the different possibilities to load external configuration files. The details can be given on the command line, or, for example, additional configuration files can be reloaded in an *application.properties* file.

Loading configuration files also can be triggered declaratively in the source code. For this, annotation `@PropertySource`[12] is added to `@Configuration`. For example, there should be another property file *email.properties* in the classpath:

```
@Configuration
@PropertySource( "classpath:email.properties" )
class …
```

This *email.properties* file is obtained in addition to the known configuration files.

---

[11] *https://docs.spring.io/spring-boot/docs/current/reference/html/howto.html*

[12] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/context/annotation/PropertySource.html*

The `@PropertySource` annotation takes an array of sources and is repeatable. If a resource file is allowed to be missing, the annotation attribute `ignoreResourceNotFound` is set to `true`. Here's an example:

```
@Configuration
@PropertySource( value = "classpath:email.properties",
                 ignoreResourceNotFound = true )
@PropertySource( value = "classpath:datasource.properties" )
class …
```

The `classpath:` prefix implies that the files *email.properties* and *datasource.properties* are obtained from the classpath, whereas *email.properties* may be missing, and *datasource.properties* must exist. In older code, you can find `@PropertySources( { @PropertySource(…), @PropertySource(…) } )` because annotation containers had to be used at that time as `@PropertySource` wasn't yet repeatable before Java 8.

Set Properties via SpringApplication[Builder]

Most of the property sources we've encountered so far are coming from external sources: either from the command line or from text documents. Configuration properties can also be set in the program code. For this, we need to turn to the `SpringApplication`[13] or `SpringApplicationBuilder`[14] class.

- **`SpringApplication`**:
  - `setDefaultProperties(Map<String,Object> defaultProperties)`
  - `setDefaultProperties(Properties defaultProperties)`
- **`SpringApplicationBuilder`**:
  - `properties(Map<String,Object> defaults)`
  - `properties(Properties defaultProperties)`
  - `properties(String... defaultProperties)`

The properties can be passed in different formats. With the `SpringApplicationBuilder` class, there is an additional method with a variable argument (vararg).

ConfigurableEnvironment

So far, we've talked exclusively about the `Environment` type, and we can inject it without any problems. There is a subtype of `Environment` called `ConfigurableEnvironment`[15] (see Figure 3.5). This is an even more interesting type because it can be used to enumerate all property sources and also add new `Environment` objects and profiles at runtime.

---

[13] *https://docs.spring.io/spring-boot/docs/current/api/org/springframework/boot/SpringApplication.html*

[14] *https://docs.spring.io/spring-boot/docs/current/api/org/springframework/boot/builder/SpringApplicationBuilder.html*

[15] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/core/env/ConfigurableEnvironment.html*

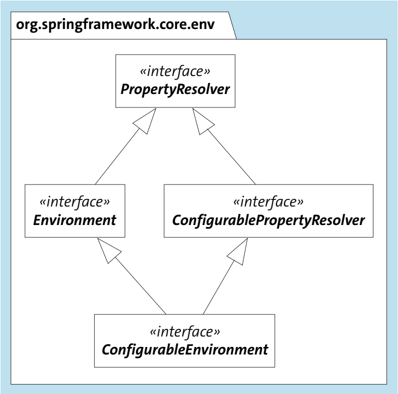

org.springframework.core.env
«interface»
PropertyResolver
«interface»
Environment
«interface»
ConfigurablePropertyResolver
«interface»
ConfigurableEnvironment

Figure 3.5 "ConfigurableEnvironment" Subtype

There are two ways to get a `ConfigurableEnvironment`:

- A `ConfigurableEnvironment` can be injected.
- The `ConfigurableApplicationContext` we get back from the `run(…)` method when starting the container provides the `getEnvironment(…)` method; it returns the special `ConfigurableEnvironment` subtype.

The `getEnvironment(…)` method at `ConfigurableApplicationContext` is declared in the `EnvironmentCapable` base type, as shown in the Unified Modeling Language (UML) diagram in Figure 3.6.

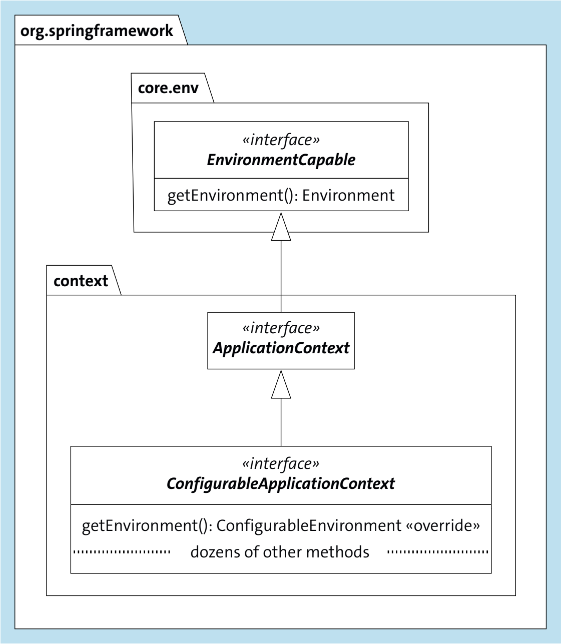

org.springframework
core.env
«interface»
EnvironmentCapable
getEnvironment(): Environment
context
«interface»
ApplicationContext
«interface»
ConfigurableApplicationContext
getEnvironment(): ConfigurableEnvironment «override»
dozens of other methods

Figure 3.6 Extended Return Type of "getEnvironment(…)" in "ConfigurableApplicationContext

For a brief demonstration, we can employ the `ConfigurableEnvironment` class, list all the property sources, and then inspect their contents:

```
configurableEnvironment.getPropertySources().forEach(
  propSource -> log.info( "{}= \n{}", propSource, propSource.getSource() )
);
```

Here are the output results:

```
ConfigurationPropertySourcesPropertySource {name='configurationProperties'}=
o.s.b.context.properties.source.SpringConfigurationPropertySources@6b04acb2

PropertiesPropertySource {name='systemProperties'}=
{java.specification.version=17, …, java.class.version=61.0}

OriginAwareSystemEnvironmentPropertySource {name='systemEnvironment'}=
{USERDOMAIN_ROAMINGPROFILE=YOGA, …, ZES_ENABLE_SYSMAN=1}

RandomValuePropertySource {name='random'}=
java.util.Random@18388a3c

OriginTrackedMapPropertySource {name='Config resource 'class path resource ↩
[application.properties]' via location 'optional:classpath:/''}= ↩
{logging.level.com.tutego=INFO, logging.level.org.springframework=ERROR, …, ↩
tutego.trainings=${tutego.homepage}/seminare}
```

This can be useful for debugging and logging. We'll see in Chapter 10, Section 10.3, that Spring Boot Actuator can also provide the information from the `Environment` via a REST endpoint.

Your Own Property Sources

The predefined property sources might not be enough if, for example, data is read from special sources with unusual file formats (think of Excel documents or comma-separated values [CSV] files) or data structures. In Spring, it's possible to also use your own property sources. So that the example doesn't become too comprehensive, we'll write a new property source for random numbers, which is strongly based on class `RandomValuePropertySource` of Spring Boot.

A custom property source will implement the `PropertySourceFactory` interface.[16] There are other alternatives, but let's concentrate on this one. The functional interface prescribes method `createPropertySource(…)` and returns `PropertySource` with the data. So, `PropertySourceFactory` is nothing more than a creator for `PropertySource` objects.

---

[16] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/core/io/support/PropertySourceFactory.html*

Here's the implementation:

```
class SecureRandomPropertySourceFactory implements PropertySourceFactory {
  private PropertySource<Object> securePropertySource =
    new PropertySource<>( "securerandom" ) {
      private final SecureRandom random = new SecureRandom();
      @Override public Object getProperty( String name ) {
        return switch ( name ) {
          case "securerandom.int"  -> random.nextInt();
          case "securerandom.long" -> random.nextLong();
          default -> null;
        };
      }
    };
  @Override
  public PropertySource<?> createPropertySource(
                             String __, EncodedResource ___ ) {
    return securePropertySource;
  }
}
```

The variable `securePropertySource` is of type `PropertySource`,[17,18] and the inner anonymous class overrides abstract method `getProperty(String)`, which returns the value of the key. Because a property source must have a name, we pass `securerandom`; however, the name isn't relevant to us.

Spring passes the property name to the `getProperty(…)` method, and we need to resolve the name. There are only two possibilities: either we supply a random number of the desired size for the name or we don't, in which case, the implementation returns `null`.

There are different ways to register `PropertySourceFactory`. A simple way—which isn't necessarily the cleanest—is to set the class via `@PropertySource`:

```
@Configuration
@PropertySource( value   = "",
                 factory = SecureRandomPropertySourceFactory.class )
class …
```

Normally, `value` contains a source, but we don't need it, and initialize the variable with an empty string. This is necessary because `value` has no default value. Variable `factory` stores the type token for the `SecureRandomPropertySourceFactory`.

As you know, it's then possible to work with `getProperty(…)` on the `Environment`:

---

[17] For those having déjà vu, there is also the annotation type `org.springframework.context.annotation.PropertySource`.

[18] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/core/env/PropertySource.html*

```
var rnd1 = environment.getProperty( "securerandom.int" );
var rnd2 = environment.getProperty( "securerandom.long" );
```

Spring Cloud Config

In our current system, `Environment` is populated by a set of property sources containing local information. This information is either hard-coded in the application files *application[properties|yml]* or provided locally on the system. However, this approach becomes impractical in a cloud environment, where making a configuration change requires triggering a new deployment.

To address this issue, the Spring team has developed an interesting project called *Spring Cloud Config*.[19] With this project, a server can provide a configuration document for a client to include, eliminating the need for local configurations. The client simply needs to add a small dependency, annotate the configuration with `@EnableConfigServer`, and set a URL.

When the Spring application starts, it connects to the config server, loads the configuration data, and inserts the properties into `Environment`. This allows the data to be redistributed back into the `@ConfigurationProperties` objects or `@Value` variables. This way, configurations can be easily changed on the server side, and the client doesn't have to be rebuilt because the configurations are physically separated from the deployed unit.

This approach provides a more efficient way of managing configurations in a cloud environment, where changes need to be made quickly and without the need for a full redeployment. It allows for easy changes to be made on the server side, which are then automatically distributed to the client applications without requiring any manual intervention.

3.2.11 Define Spring Profiles

Spring provides a powerful feature called *profiles*, which allows for the loading of different sets of configuration properties depending on the current environment. This feature is essential for creating flexible and adaptable applications that can function differently in various environments.

Each profile is identified by a unique name, and there are no predefined names for them. The name of the profile can be customized according to the specific requirements of the application. Some commonly used profile names include “dev” for development, “test” for testing, “staging” for staging environments, “production” for production environments, and even more specific profile names such as “database-customer-acme-corp” or “messaging-wonka-industries” based on the needs of the application.

---

[19] *https://cloud.spring.io/spring-cloud-config/reference/html/*

Files for Spring Profiles

Profile-specific information is usually stored in files. The file names have a special structure: they start as usual with *application*, followed by a minus sign and the name of the profile and finally, depending on the format, the file extension *.properties* or *.yml*. Profile files are searched in the same place, where the *application.[properties|yml]* files are also searched.

An example with three profile files called *cloud*, *uat*, and *demo*:

- *src*
  - *main*
    - *java*
  - ...
  - *resources*
    - *application.properties*
    - *application-cloud.properties*
    - *application-uat.properties*
    - *application-demo.yml*

Loading Sequence of the Spring Profiles

If a Spring application starts, then the configurations are loaded in a specific order:

1. The *application.[properties|yml]* files are loaded.
2. The profile data is read.

This profile data might overwrite the properties from *application.[properties|yml]* because the data in the profiles have a higher priority. However, as far as priority is concerned, there are other property sources that have higher and lower priority than the *application*.[properties|yml]* files. The order in the list of property sources described in Section 3.2.10 also indicates the priority.

Default Profile

By default, all properties in *application.[properties|yml]* don't belong to any profile and are always loaded. If *no* profile is enabled, Spring searches for optional *application-default.[properties|yml]* files and loads them; the files belong to the *default profile*. However, if a profile is set, the *application-default.[properties|yml]* files aren't loaded, so the default profile is ignored.

Multi-Document YAML Files

For each profile for configuration, there will be corresponding *.properties* or *.yml* files. This could quickly become confusing. Therefore, for *.yml* and *.properties* files, Spring supports *multi-document files*. With their help, multiple documents can be combined into one document.

To begin, consider a YAML file that contains multiple documents:

```
date4u:
  filesystem:
    minimum-free-disk-space: 1000000
---
spring:
  config:
    activate:
      on-profile: dev
date4u:
  filesystem:
    minimum-free-disk-space: 100000
---
spring:
  config.activate.on-profile: test
date4u:
  filesystem.minimum-free-disk-space: 10000
```

The YAML document is composed of three documents separated by `---` (three minus signs[20]). This is useful when you want to maintain different profile documents in only one file.

Our example shows that `minimum-free-disk-space` is set to `1000000` by default. The following configuration properties should only be set if the application was started in a certain profile. To specify that the following properties should only apply to a certain profile, property `spring.config.activate.on-profile` is prepended after `---`, and the properties that only apply to the named profile follow. If the profile `dev` was set, then later `date4u.filesystem.minimum-free-disk-space` is set to `100000`. It's important to note that `spring.config.activate.on-profile` doesn't set a profile, but is only a kind of marker for the following configurations, which then belong to the profile. So, for the profile `test` the value of `minimum-free-disk-space` is set to `10000`.

As shown in the YAML file, a dot can also be used as a separator so that the information can be written more compactly—so a new line isn't necessary for each additional depth. The following three notations are identical:

```
date4u:
  filesystem:
    minimum-free-disk-space: 20000
```

and

---

[20] *https://yaml.org/spec/1.2.2/#22-structures*

```
date4u:
  filesystem.minimum-free-disk-space: 20000
```

and

```
date4u.filesystem.minimum-free-disk-space: 20000
```

Already, a YAML file looks like a properties file.

Multi-Document Properties Files

The creators of Spring Boot found multi-document files for YAML files so interesting that they wanted to transfer the feature to regular property files. However, property files don't support multiple documents at all, so the Spring team developed their own syntax and parser: first, the comment character is set for property files (#, number sign), and then `---` (three minus signs) follow. This special extended properties format is called *multi-document properties file*. In short, `#---` separates the documents.

To illustrate this, let's look at an example:

```
date4u.filesystem.minimum-free-disk-space=1000000
#---
spring.config.activate.on-profile=dev
date4u.filesystem.minimum-free-disk-space=100000
#---
spring.config.activate.on-profile=test
date4u.filesystem.minimum-free-disk-space=10000
```

Property `minimum-free-disk-space` is still set to `1000000`. If the `dev` profile is set, `100000` bytes of free memory should apply. For this, as with the YAML files, assignment `spring.config.activate.on-profile=dev` is located after `#---`, which expresses that the following configurations only apply in the `dev` profile. In profile `test`, property `minimum-free-disk-space` is then `10000`.

[+]

Tip

The alternative to multi-document files are different profile files, and that could become confusing. The advantage of multi-document files is that all information can be found in one place. This makes it much faster for us as humans to read the context because we just have to look up or down a few lines. This can be easier than opening and comparing different files.

Ultimately, it's a matter of taste. The advantage of single files is that when a property of a profile is changed, only a smaller file needs to be modified; the change tracking in version management makes this easy to track. In contrast, with a multi-document file, we would have more frequent changes to a large file if we wanted to change details about a specific profile. Changes then show up permanently in version management, although they may only be local changes to a single profile.

3.2.12 Activate Profiles

A profile must be consciously activated for use. Configuration property `spring.profiles.active` is used for this. It can be set like any other property; here are three examples:

- Set via the command line:

```
--spring.profiles.active=uat
```

- Set via *application.properties*:

```
spring.profiles.active=uat
```

- Set via *application.yml*:

```
spring:
    profiles:
        active: "uat"
```

The profile can also be set via the application context:

```
applicationContext.getEnvironment().setActiveProfiles( "uat" );
```

Task: Set Profile

If you want to practice what you've learned, implement the following:

- Crate a profile `test` that sets the debug level for `com.tutego.date4u` to `DEBUG`.
- Write a profile `dev` that sets the debug level for `com.tutego.date4u` to `TRACE`.

Remember, by default, the log level is set to `INFO`.

Proposed solution: First, two new files are created, as shown in the following listings.

```
logging.level.com.tutego.date4u=DEBUG
```

Listing 3.6 src/main/resources/application-test.properties

```
logging.level.com.tutego.date4u=TRACE
```

Listing 3.7 src/main/resources/application-dev.properties

If there are log calls in a class in a package below `com.tutego.date4u` such as

```
log.debug( "Debug" );
log.trace( "Trace" );
```

then you'll won't see any output without the log level set.

For example, if you set the log level in *application.properties* as

```
spring.profiles.active=dev
```

then you'll see all log outputs.

Enable Multiple Profiles

The examples have exactly one profile activated, but multiple profiles can be activated. This is especially useful if you want to combine different technologies, for example, for “Database Development” or “Database Productive,” “Messaging System Development” or “Messaging System Productive,” “Email Server Development,” and “Email Server Test.” It’s not necessary to combine everything that belongs to the test phase or to the development phase in one profile, for example, but it can be more finely granulated.

Multiple profiles are separated with a comma when you use `spring.profiles.active`. Here’s an example:

```
--spring.profiles.active=uat,noopdatastore
```

With `setActiveProfiles(…)`, multiple profile names can be set because the parameter is a vararg:

```
applicationContext.getEnvironment().setActiveProfiles("uat, noopdatastore");
```

Profiles can be combined into a profile group—the reference documentation explains how to do this.[21] This is handy to combine certain combinations, such as “Database customer acme corp” and “Messaging system customer acme corp.”

Spring Framework can include another profile using `spring.profiles.include`. The reference documentation explains this in more detail.[22]

3.2.13 Spring-Managed Beans Depending on Profile

If a profile is activated, the Spring Framework will automatically load the appropriate files. However, Spring-managed beans can also be built depending on a profile. For this purpose, annotation `@Profile`[23] is used to set profile names. The annotation is placed at the following:

- `@Component`
- `@Configuration`
- `@Bean` methods specifically

If the container starts up, and the profile is as specified at `@Profile`, new Spring-managed beans are created; otherwise, they aren’t.

---

[21] *https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#features.profiles.groups*

[22] *https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#features.profiles.adding-active-profiles*

[23] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/context/annotation/Profile.html*

Note: Background

The `@Profile` annotation is so far the only `@Conditional` provided by the Spring Framework:

```
@Target(value={TYPE,METHOD})
 @Retention(value=RUNTIME)
 @Documented
@Conditional(
value=org.springframework.context.annotation.ProfileCondition.class)
public @interface Profile
```

That is, `ProfileCondition` effectively controls whether a bean should be instantiated or not.

To demonstrate this, we'll examine two examples that use factory methods:

```
@Configuration
public class AppConfig {
  @Bean @Profile( "dev" )
  public DataSource devDataSource() { … }

  @Bean @Profile( "production" )  // @Profile( "!dev" )
  public DataSource prodDataSource() { … }
}
```

The factory methods create certain `DataSource` instances, that is, database connections, depending on the `dev` or `production` profile. In other words, if `dev` is active, then this condition "matches," and the factory method `devDataSource()` is called and returns a new `DataSource`. It's different with `production`: if the profile is set, the `prodDataSource()` method is called.

With two profiles, `dev` and `production`, the use of `@Profile( "!dev" )` is equivalent to `@Profile( "production" )`. The exclamation mark is part of a *profile expression*.

Profile Expressions

The `@Profile` annotation allows setting not only a single profile name but also multiple profile names that can be combined as well. Here are some examples to illustrate this. Suppose there are three profiles named `p1`, `p2`, `p3`:

- **`@Profile( "!p1" ) // not p1`**

  The Bean is generated when the profile `p1` isn't active.

- **`@Profile( {"p1", "p2"} ) // p1 or p2`**

  An array of profile names can be specified. In this example, it means that either profile `p1` or profile `p2` is active.

- **`@Profile( {"p1", "!p2"} ) // p1 active, or p2 not active`**
  Individual specifications can be negated; in the example, the profile `p1` is activated or not `p2`.

Further combinations with `&` (and) and `|` (or) are possible and maybe even more readable than the information in the array:

- **`@Profile( "p1 & p2" )`**
  Profiles `p1` *and* `p2` must be active.
- **`@Profile( "p1 | p2" )`**
  Profiles `p1` *or* `p2` must be active.

Combinations of logical operations are also possible:

```
@Profile( "(p1 & p2) | p3" ) // Brackets are necessary
```

If you use the operators `&` and `|` at the same time, parentheses are necessary for the priority.

3.3 At the Beginning and End

There are several ways in which a Spring Boot program can execute code after the container is started:

1. The container sends out events at certain times, and there is a special program-started event among them. If our program listens for this event with a listener, our code can be executed after the start. We look at events again in more detail in Section 3.4.
2. After building the entire object graph, a custom bean can be fetched from the context, and a custom start method can be called. We've seen how to fetch beans from the context earlier.
3. Spring Boot can execute runners `ApplicationRunner` and `CommandLineRunner`.

3.3.1 CommandLineRunner and ApplicationRunner

After the container is initialized, Spring Boot executes two *runners*: `ApplicationRunner` and `CommandLineRunner` (see Figure 3.7). They are primarily made for command-line applications, where Spring Boot passes the command-line arguments to the runners.

These two types are typical of Spring Boot. Much of what we've seen so far has been from the Spring Framework, but these types are found in Spring Boot—which is very fitting, as Spring Boot simplifies the start of the application.

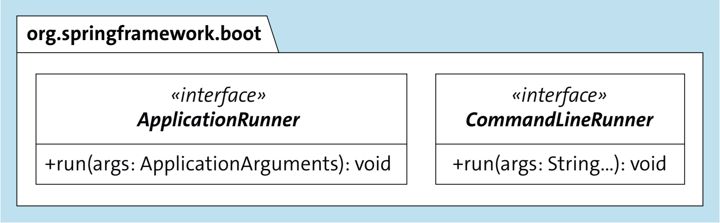


Figure 3.7 Spring Boot Interfaces "ApplicationRunner" and "CommandLineRunner"

The two (functional) interfaces have a `run(…)` method without any return, can both throw arbitrary exceptions, and differ only in the parameter type:

- `ApplicationRunner`: `void run(ApplicationArguments args) throws Exception`
- `CommandLineRunner`: `void run(String... args) throws Exception`

Once the parameter type is `ApplicationArguments`, and once it's a vararg of strings. The difference is that `CommandLineRunner` gets the `args` from `main(String[] args)` more or less directly, while in `ApplicationRunner`, Spring Boot converts the passed parameters and options into an `ApplicationArguments` object, so the console parameters are easier for us to process.

ApplicationArguments

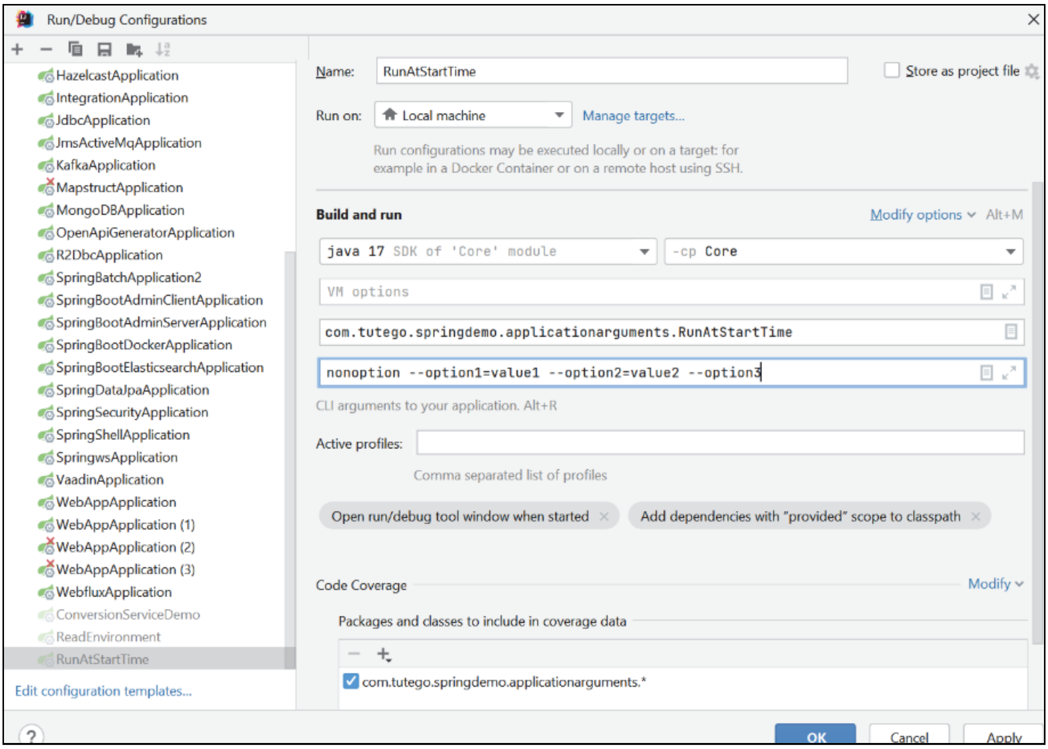
The `ApplicationArguments` class declares the following methods:

- `boolean containsOption(String name)`
- `List<String> getNonOptionArgs()`
- `Set<String> getOptionNames()`
- `List<String> getOptionValues(String name)`
- `String[] getSourceArgs()`

An `ApplicationRunner` could look like this:

```
@Component
class RunAtStartTime implements ApplicationRunner {
    @Override
    public void run(ApplicationArguments args) {
        // nonoption --option1=value1 --option2=value2 --option3
        System.out.println(args.getNonOptionArgs());
        System.out.println(args.getOptionNames());
        System.out.println(Arrays.toString(args.getSourceArgs()));
        for (String optionName : args.getOptionNames()) {
            System.out.println(args.getOptionValues(optionName));
        }
    }
}
```

The `RunAtStartTime` class object becomes a Spring-managed bean through `@Component` so that the runner can also be found and executed after the container starts up.


Run/Debug Configurations
HazelcastApplication
IntegrationApplication
JdbcApplication
JmsActiveMqApplication
KafkaApplication
MapstructApplication
MongoDBApplication
OpenApiGeneratorApplication
R2DbcApplication
SpringBatchApplication2
SpringBootAdminClientApplication
SpringBootAdminServerApplication
SpringBootDockerApplication
SpringBootElasticsearchApplication
SpringDataJpaApplication
SpringSecurityApplication
SpringShellApplication
SpringwsApplication
VaadinApplication
WebAppApplication
WebAppApplication (1)
WebAppApplication (2)
WebAppApplication (3)
WebfluxApplication
ConversionServiceDemo
ReadEnvironment
RunAtStartTime
Edit configuration templates...
Store as project file
Manage targets...
Modify options Alt+M
Modify
OK
Cancel
Apply

Name: RunAtStartTime

☐ Store as project file

Run on: Local machine

Run configurations may be executed locally or on a target: for example in a Docker Container or on a remote host using SSH.

Build and run

java 17 SDK of 'Core' module | -cp Core

VM options

com.tutego.springdemo.applicationarguments.RunAtStartTime

nonoption --option1=value1 --option2=value2 --option3

CLI arguments to your application. Alt+R

Active profiles:

Comma separated list of profiles

Open run/debug tool window when started | Add dependencies with "provided" scope to classpath

Code Coverage

Packages and classes to include in coverage data

☑ com.tutego.springdemo.applicationarguments.*

Figure 3.8 Arguments Set in the Integrated Development Environment (IDE)

Suppose the program is started on the command line with the argument `nonoption --option1=value1 --option2=value2 --option3`. Then the program output is as follows:

```
[nonoption]
[option3, option1, option2]
[nonoption, --option1=value1, --option2=value2, --option3]
[]
[value1]
[value2]
```

Note

`ApplicationArguments` can also be injected, and `ApplicationRunner` wouldn't be necessary.

In principle, several runners are allowed, whereby the order is then sometimes relevant. There is a possibility to determine this via `@Order`—you can find more information about this in Chapter 2, Section 2.7.5.

3.3.2 At the End of the Application

If the Spring program doesn't start any special background threads or if it doesn't wait for incoming data, the program terminates automatically. Typically, enterprise applications run in a loop and must shut down correctly. For example, data in the buffer must be written, resources must be released, and transactions must be completed.

Shutting Down the Application

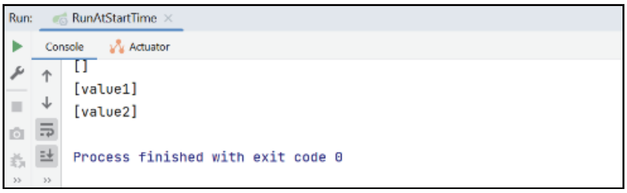
To shut down a Spring Boot application in a controlled manner, the `close()` method is called on `ConfigurableApplicationContext`—the context is `AutoCloseable`. We can inject the context and then call `close()` later in the code. We'll see how the wiring works later.[24]

Closing triggers a `ContextClosedEvent` to which custom programs can respond. We'll look at listening for events in more detail in the next Section 3.4.

Exit Code

When a program *exits*, whether it's a Java program or another program, it returns an *exit code* at the end of the program. The exit code is—and this depends on the operating system—generally an integer and lies between 0 and 255. If numbers are larger, the remainder operator 256 is usually applied to them, so that the range of values again lies between 0 and 255. Incidentally, the same applies to negative numbers, which thus basically also just become positive numbers. There could be differences in the implementations and in the operating systems.

Exit code 0 represents a successful shutdown (called `EXIT_SUCCESS`), and everything else that isn't equal to 0 represents an error. This exit code allows you to control the flow of the batch program, for instance, if you call the Java program in a shell. This is useful because it allows you to control the following operations depending on states: if the Java program could not do certain things, then subsequent programs dependent on it shouldn't run either. If you work under the Windows terminal, you'll find the exit code also under the term *Errorlevel*.



```
[]
[value1]
[value2]

Process finished with exit code 0
```

Figure 3.9 Exit Code in the IDE

---

[24] In the code, it will look something like this later: with `@Autowired ConfigurableApplicationContext context;`, Spring will give us the context object, and we can later close the application with `context.close()`.

Java applications return 0 by default if the program terminated correctly. If an uncaught unchecked exception arrives at the JVM, error code 1 is automatically returned. This can also be seen in the terminal window of an IDE (see Figure 3.9).

3.3.3 Exit Java Programs with Exit Code *

A Java program can exit the JVM at any time and return an exit code. Static method `System.exit(int)` is used for this purpose; the exit code is passed as an integer. This `exit(…)` method does two things:

- Terminates a Java program
- Gives an exit code to the caller

We can use this call to `System.exit(int)` in our Spring Boot program in principle, but Spring uses a special data type to determine the exit code, so you can separate the determination of the exit code from your own program end.

ExitCodeGenerator, SpringApplication.exit(…)

Spring Boot has functional interface `ExitCodeGenerator` that declares method `getExitCode()`. We can implement this exit code generator, and it will tell what exit code to return.

Spring considers the return value of `ExitCodeGenerator` implementations in two places: first, when a Spring-managed bean implements the `ExitCodeGenerator` interface, and, second, when an instance is passed to a special `exit(…)` method. `SpringApplication` declares the static method `int exit(ApplicationContext context, ExitCodeGenerator... exitCodeGenerators)`, which does the following:

- Runs and iterates over all parameters and beans of type `ExitCodeGenerator` and determines the final exit code (our own `ExitCodeGenerator` implementations can be ordered)
- Sends an event in case of an error (i.e., if the exit code isn't 0) (a listener for an `ExitCodeEvent` can be implemented and asked what this exit code was)
- Closes the context
- Returns the exit code

The first parameter type of `exit(ApplicationContext, ExitCodeGenerator...)` is `ApplicationContext` because this type declares the method to close the context with `close()`. This doesn't sound very object-oriented, and you might rather assume the `exit(…)` method to be `ApplicationContext`. But `ExitCodeGenerator` is a type from Spring Boot, just like `SpringApplication`; `ApplicationContext`, however, originates from the Spring Framework, and no type relationship can be drawn to Spring Boot.

It's important to understand that `SpringApplication.exit(…)` doesn't call `System.exit(…)` in the background, but only gets the status code from all exit code generators, sends an event in case of an error, and closes the context. If we then really want to close the application, we'll have to use `System.exit(…)` ourselves.

The parameter list of the `exit(ApplicationContext context, ExitCodeGenerator... exitCodeGenerators)` method shows that a variable number of `ExitCodeGenerator` instances can be passed. Variable means that not necessarily an `ExitCodeGenerator` must be passed. This is because Spring-managed beans in the context that implement `ExitCodeGenerator` are also recognized.

To provide a clearer understanding, let's examine an example that includes two classes: on one side, we have the demo program, and, on the other side, we have a Spring-managed bean that implements functional interface `ExitCodeGenerator`:

```
@SpringBootApplication
public class ExitCodeGeneratorDemo {
  public static void main( String[] args ) {
    var ctx = SpringApplication.run( ExitCodeGeneratorDemo.class, args );
    System.exit( SpringApplication.exit( ctx ) );
  }
}

@Component
class MyExitCode implements ExitCodeGenerator {
  @Override public int getExitCode() {
    return 2;
  }
}
```

The two calls of `exit(…)` can be seen well. The inner call determines the exit code from the recognized exit code generators, and the result is used in the `System.exit(…)` method because only `System.exit(…)` will terminate the JVM. Although `SpringApplication.exit(…)` would also shut down the container in the best case via closing the context, `System.exit(…)` is ultimately necessary for returning the status code to the JVM and operating system.

In some cases, the program encounters errors that prevent it from continuing and must be terminated. Here's another example to illustrate this scenario: suppose a program encounters a fictional condition (represented here in pseudocode); in this case, it should be terminated with an exit code of `2`:

```
@Component
class ValidOrExit implements CommandLineRunner {

  @Autowired private ApplicationContext ctx;

  @Override public void run( String... args ) {
    if ( condition )
      System.exit( SpringApplication.exit( ctx, () -> 2 ) );
  }
}
```

To use `SpringApplication.exit(…)`, we always need `ApplicationContext`. Outside of the `main(…)` method, we have to get that somehow, but that's easy with injection; we just "wish" for the reference to the current `ApplicationContext`.

The `Application.exit(…)` method can then be passed the `ApplicationContext` plus a piece of code that returns the exit code for the error case. The compact notation uses a lambda expression that returns `2`, which is also what `System.exit(…)` gets.

There are three reasons not to use a simple `System.exit(2)`:

1. Another `ExitCodeGenerator` may have priority over our lambda expression that we pass to `exit(…)` via the vararg. That is, another `ExitCodeGenerator` could be very high priority and return a completely different exit code.
2. `Application.exit(…)` triggers an event when an exit code other than `0` occurs.
3. The context is shut down, which results in the application shutting down. `System.exit(…)` exits the application quite hard and is intended for the JVM to return the exit code, but not to fire the context in a controlled manner.

Implement Exception Classes ExitCodeGenerator

The use of `System.exit(…)` and `SpringApplication.exit(…)` is programmed very explicitly because a program must first reach a place with the `exit(…)` call.

There is another way to translate an exception into exit code. Unchecked exceptions arrive automatically at the framework in many places, and Spring calls the `getExitCode()` method on exception classes that implement `ExitCodeGenerator`[25] and automatically passes the determined value out via `System.exit(…)`:

```
class InsufficientDiskSpaceException extends RuntimeException
                                     implements ExitCodeGenerator {
  InsufficientDiskSpaceException( String message ) {
    super( message );
  }

  @Override
  public int getExitCode() {
    return 2;
  }
}
```

---

[25] *https://docs.spring.io/spring-boot/docs/current/api/org/springframework/boot/ExitCodeGenerator.html*

Elsewhere it may say the following:

```
throw new InsufficientDiskSpaceException( “Was only 2 KiB” );
```

With such a construction, we save the `SpringApplication.exit(…)` and also the `System.exit(…)`.

The disadvantage of this solution is that two concepts are mixed up, which isn’t in good object-oriented style: An exception is mixed up with a reference to the Spring Framework with the `ExitCodeGenerator`. An exception is now associated with an exit code, and it’s questionable what an `InsufficientDiskSpaceException` has to do with an exit code used at the end of the program.

ExitCodeExceptionMapper

Spring Boot provides a way to separate the exception from the exit code. A Spring-managed bean can implement the `ExitCodeExceptionMapper` functional interface[26] and perform a translation of the exception to the exit code in the `getExitCode(Throwable exception)` method:

```
@Component
class MyExitCodeExceptionMapper implements ExitCodeExceptionMapper {
  @Override public int getExitCode( Throwable exception ) {
  return
   (exception.getCause() instanceof InsufficientDiskSpaceException) ? 2 : 0;
  }
}
class InsufficientDiskSpaceException extends RuntimeException {
  InsufficientDiskSpaceException( String message ) {
    super( message );
  }
}
```

In the current implementation, the `Throwable` that Spring passes into `getExitCode(…)` isn’t the exception that was thrown, but an `IllegalStateException`. Before the mapper checks the exception types, that exception is retrieved via `getCause()`, and the reason can be checked. In our example, we return `2` if the exception was an `InsufficientDiskSpaceException`, otherwise, we return `0`. Whether `0` is the best return is another story because, with `0`, we actually say that everything was fine.

ApplicationPid[FileWriter]

An application is best terminated from the inside via a `close()` on the context. That is, for example, if there is an indication from outside that the container should be shut down, we can call `close()` or use `SpringApplication.exit(…)`, which leads to the `close()` from the context.

---

[26] *https://docs.spring.io/spring-boot/docs/current/api/org/springframework/boot/ExitCodeExceptionMapper.html*

In principle, Java programs can be "killed" from the outside, and this may be necessary and intentional. But to be able to kill a program from the outside, a *process identifier (PID)* is necessary. Java Platform, Standard Edition (Java SE) can determine its own process identifier, and Spring Boot provides two interesting helper classes to complement it: `ApplicationPid` and `ApplicationPidFileWriter`.

`ApplicationPid`[27] represents a PID, and `ApplicationPidFileWriter`[28] is an `ApplicationListener<SpringApplicationEvent>` that automatically writes a PID to a file after startup. Listeners are automatically processed by Spring in the lifecycle. See the Section 3.4 for more on this.

Here is a brief demonstration program to examine:

```
@SpringBootApplication
public class ShutdownDemo {
  public static void main( String[] args ) {
    SpringApplication app = new SpringApplication( ShutdownDemo.class );
    app.setDefaultProperties( Map.of("spring.pid.fail-on-write-error",true) );
    app.addListeners( new ApplicationPidFileWriter() ); // "application.pid"
    try ( ConfigurableApplicationContext ctx = app.run( args ) ) {
      Logger log = LoggerFactory.getLogger( ShutdownDemo.class );
      log.info( "{}", new ApplicationPid() );
      new Scanner( System.in ).nextLine();
    }
  }
}
```

This time the application isn't started with the static `run(…)` method, but uses the `SpringApplication` class as an instance to add this particular `ApplicationPidFileWriter` with `addListeners(…)`. (In principle, this could have been done with the listener using a builder, but this solution is short and sweet.)

The `ApplicationPidFileWriter` can be further configured. For example, if the file system is read-only, the exceptions that occur when a write is attempted are automatically suppressed. This can be changed via a property, and therefore the program sets `spring.pid.fail-on-write-error` to `true`. `ApplicationPidFileWriter` has a parameterless constructor that leaves the file name for the PID at *application.pid*, or a parameterized constructor that allows the file name to be changed.

---

[27] *https://docs.spring.io/spring-boot/docs/current/api/org/springframework/boot/system/ApplicationPid.html*

[28] *https://docs.spring.io/spring-boot/docs/current/api/org/springframework/boot/context/ApplicationPidFileWriter.html*

Once the `SpringApplication` object is built, the container can be started. However, because the PID file is automatically deleted after the end, the program waits for user input so that we can study the PID file.

Applications can be terminated externally via the PID. Under Unix, this is done with `kill <PID>`; this is identical to `kill -15 <PID>` (SIGTERM signal) or "hard" with `kill -9 <PID>` (SIGKILL signal):

```
$ cat application.pid | xargs kill -9
```

With SIGTERM, the shutdown hooks run; a SIGKIL can no longer process a program, and the operating system terminates the program.

3.4 Event Handling

One of the essential features that Spring provides is the ability to respond to and broadcast events. This decouples the sender and receiver of the events, allowing for more flexible and modular programming.

COFFEE!
SPAGHETTI!

3.4.1 Participating Objects

The event handling feature in Spring brings together three objects: the event source, the listener, and the event itself. These objects are described here:

- **Event source**
  This object generates the event. It could be a user interface element, a database, or any other object that produces events.
- **Listener**
  Also known as the event handler, this object receives the event and performs some action. It's responsible for handling the event and doing something based on the event's data.

- **Event**
  The object itself is transported from the event source to the listener as an object. The event object contains information about the event, such as its type and data.

In Spring, events are an integral part of the framework and can be used to trigger a wide range of actions. They are often used to decouple different components of an application, allowing them to interact in a more flexible and modular way. The ability to handle events is also essential for creating reactive applications that respond to changes in real time.

3.4.2 Context Events

Spring itself fires a few events, for example, when something happens in the container. By default, we have four event types:

- `ContextRefreshedEvent`
  Sent when the `ApplicationContext` is initialized or refreshed, that is, when `context.refresh()` is called.
- `ContextStartedEvent`
  Sent when the `ApplicationContext` is started by calling `context.start()`.
- `ContextStoppedEvent`
  Sent when the `ApplicationContext` is stopped with `context.stop()`.
- `ContextClosedEvent`
  Sent when `ApplicationContext` is closed with the `context.close()` method.

The listed events are triggered by the container. Furthermore, there are other events from special Spring packages, such as the `org.springframework.web.context.support.RequestHandledEvent` from the Spring Web MVC framework, when a request is completed. The `ServletRequestHandledEvent` event type is a subtype of this. However, this event type isn't relevant for us, and it's only listed here as an example that it doesn't have to stay with the mentioned types.

Type Relationships of the Event Classes

Spring's events are related by type, as shown in the UML diagram in Figure 3.10.

At the center is the type `ApplicationEvent`, which is a subclass of `java.util.EventObject`, which itself is a serializable object. `EventObject` is relatively unspectacular and irrelevant in practice when writing your own event classes.

`ApplicationEvent` originates from the `org.springframework.context` package, and for this abstract base class, there are a number of subclasses, for example, `ApplicationContextEvent`, which we just discussed. `ApplicationContextEvent` is the abstract base class for events raised by the Spring container. There aren't so many event classes from any Spring Boot package. One example is the `ExitCodeEvent` class; this event is fired exactly when the application is terminated, and then the exit code can be taken from this event. An example of the `ExitCodeEvent` is shown in Section 3.4.5

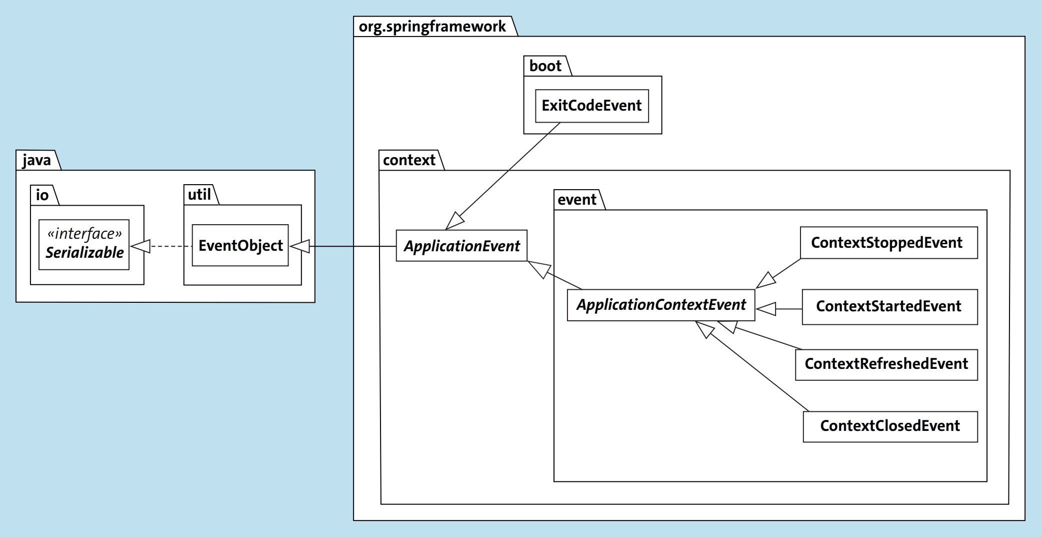

org.springframework
boot
ExitCodeEvent
context
event
java
io
util
«interface»
Serializable
EventObject
ApplicationEvent
ContextStoppedEvent
ApplicationContextEvent
ContextStartedEvent
ContextRefreshedEvent
ContextClosedEvent

Figure 3.10 Type Relationships of the Central Spring Events

3.4.3 ApplicationListener

There are only two ways to program a listener and declare a callback method. The first way is to implement a special Spring interface called `ApplicationListener`:[29]

```
@FunctionalInterface
public interface ApplicationListener<E extends ApplicationEvent>
    extends EventListener {

  void onApplicationEvent(E event);

  static <T> ApplicationListener<PayloadApplicationEvent<T>>
  forPayload(Consumer<T> consumer) {
    return event -> consumer.accept(event.getPayload());
  }
}
```

`ApplicationListener` is a generic type, and a type argument must be set, which is exactly the event type you want Spring Framework to respond to.

When implementing functional interface `ApplicationListener`, method `onApplicationEvent(…)` must be implemented. Here's an example:

---

[29] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/context/ApplicationListener.html*

```
@Component
public class MyListener
implements ApplicationListener<ContextStartedEvent> {
  public void onApplicationEvent( ContextStartedEvent e ) {
    …
  }
}
```

The generic type argument, here `ContextStartedEvent`, also becomes the parameter type. The class is a regular `@Component` that is detected via component scanning. Spring thus knows that this component implements a listener. If a `ContextStartedEvent` occurs, then the `onApplicationEvent(…)` method is called at `MyListener`. The disadvantage of this variant is that it relies on a Spring-specific interface. This is invasive, and we didn't really want to do this: We shouldn't have Spring-specific interfaces in our source code. What is accepted, however, are annotations. That's why there is an alternative.

3.4.4 @EventListener

Annotation `EventListener`[30] provides a second option for a listener. We put this annotation on the exact methods that should be called when there is an event. To illustrate, consider the example of the `ContextStartedEvent`:

```
@Component
public class MyListener {
  @EventListener
  public void onContextStartedEvent( ContextStartedEvent e ) {
    …
  }
}
```

The parameter type determines which event type should be reacted to. In my case, this is the method `onContextStartedEvent(…)`, but the name of the method is arbitrary; you don't have to implement an interface that dictates anything.

One @EventListener for Multiple Events

It's possible to use the same listener for multiple event types. It's not necessary to write different methods, each accepting different event objects:

```
@EventListener( /*classes =*/ { ContextStartedEvent.class,
                                ContextStoppedEvent.class } )
public void onContextStartedOrStoppedEvent( ApplicationEvent event ) {
  …
}
```

---

[30] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/context/event/EventListener.html*

The example shows that with annotation attribute `classes` (an alias to `value`), an array of `Class` objects can be passed, which stand for the respective event type. Because the method can be parameterized with two different event types at the same time, the base type is used. In the example, one method responds to `ContextStartedEvent` and `ContextStoppedEvent`; the common parent type is `ApplicationEvent`.

3.4.5 Methods of the Event Classes

Through inheritance of these event classes, different methods are available in the subclasses, just as `java.lang.Object` inherits methods to the subclasses:

- In Spring, every event object is derived from `java.util.EventObject`. From there, there is only one method, `Object getSource()`, which returns the trigger of the event.
- In the `ApplicationEvent` subclass, a single additional method `long getTimestamp()` is added; it can be used to query when the event occurred. The specification is usually in milliseconds Unix time.
- The `ApplicationContextEvents` provide us information about when the container was started, when it was refreshed, and so on. We don't have an additional new method, just method `getApplicationcontext()`, which returns an `ApplicationContext` object. The implementation is `(ApplicationContext) getSource()`—so basically these are two methods with the same functionality.

Special subclasses can add more methods. As an example, we'll use the `ExitCodeEvent` class, a special `ApplicationEvent`. The additional method is `getExitCode()`. Other events, of course, don't have this method. A listener could look like this:

```
@Component
class MyExitCodeListener {
  @EventListener
  public void exitEvent( ExitCodeEvent event ) {
    int exitCode = event.getExitCode();
    …
  }
}
```

The `MyExitCodeListener` listens for an `ExitCodeEvent` and can extract the exit code with `getExitCode()`.

Just as the Spring Framework has some prebuilt event types, we too can easily write event classes, raise events, and react to those events.

3.4.6 Write Event Classes and React to the Events

In Spring, any object can be dispatched as an event. To illustrate, consider an event that should be triggered when a new photo is saved in the system. This event should contain the file name and the creation time of the photo.

```
record NewPhotoEvent( String name, OffsetDateTime dateTime ) { }
```

Listing 3.8 NewPhotoEvent.java

With a record, this is nice and tight. If we had written this as a class, we could still have derived from `java.util.EventObject`, but there is no advantage to doing so.

A listener for the `NewPhotoEvent` is written exactly as we saw before, with `@EventListener`:

```
@Service
public class Statistic {

  private final Logger log = LoggerFactory.getLogger( getClass() );

  @EventListener
  public void onNewPhotoEvent( NewPhotoEvent event ) {
    log.info( "New photo: {}", event );
  }

}
```

3.4.7 An Event Bus of Type ApplicationEventPublisher

Listening to an event is only one side, but sending an event is another. If we want to send events, we need access to an *event bus*. This event bus is of type `ApplicationEventPublisher` in Spring.[31] This is an interface that has two methods in the current version:

```
package org.springframework.context;

@FunctionalInterface
public interface ApplicationEventPublisher {

  default void publishEvent(ApplicationEvent event) {
    publishEvent((Object) event);
  }

  // @since 4.2
  void publishEvent(Object event);
}
```

The methods are called in the same way and differ only in the parameter signature. There is the original method of Spring, which sends an `ApplicationEvent`, and a somewhat

---

[31] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/context/ApplicationEventPublisher.html*

"newer" method because, since Spring Framework version 4.2, events of arbitrary types can also be sent.

From today's point of view, the type name `ApplicationEventPublisher` is also not so appropriate because the data type "publishes" not only `ApplicationEvents` but also all possible objects.

Spring also provides an implementation of this interface, and we can have an `ApplicationEventPublisher` injected. How this works exactly is shown in a small example.

```
@Service
public class PhotoService {

  @Autowired
  private ApplicationEventPublisher publisher;

  …

  public String upload( byte[] imageBytes ) {
    String imageName = UUID.randomUUID().toString();
    …
    NewPhotoEvent newPhotoEvent = new NewPhotoEvent( imageName,
                                                     OffsetDateTime.now() );
    publisher.publishEvent( newPhotoEvent );
  }
}
```

Listing 3.9 PhotoService.java Extension

The `PhotoService` has an `upload(…)` method, and this should fire an event after saving to the file system. The event object is built and sent to the `ApplicationEventPublisher`, which we can inject. This informs all the listeners that are interested in this event type.

3.4.8 Generic Events Using the PayloadApplicationEvent Example

The Spring Framework supports generic events, which eliminates the need to declare a separate data type for each event. Instead, we can use a single data type and model various events with different generic type assignments. Here's an example that uses a hypothetical `EntityUpdateEvent`:

- `EntityUpdateEvent<Profile>`
- `EntityUpdateEvent<Photo>`

Once the type argument is `Profile`, then `Photo` is used. The advantage of such generic events is that the class is typed, but this reduces the number of event classes. With the generic class, the focus is more on the payload and not so generally on the event type. The type is always an `EntityUpdateEvent`. With these generic types, you don't have to write special classes such as `EntityUpdateEventProfile` or `EntityUpdateEventPhoto`.

PayloadApplicationEvent

We could write generic classes ourselves, but Spring brings its own generic event class, and that is `PayloadApplicationEvent`.[32] Spring itself also uses the data type internally, and that is when you want to send any object that isn't an `ApplicationEvent`. Then, the `Object` is automatically wrapped in a `PayloadApplicationEvent`.

Examining the source code of `PayloadApplicationEvent` can provide valuable insights into the process of creating your own generic event classes:

```
public class PayloadApplicationEvent<T> extends ApplicationEvent
                                  implements ResolvableTypeProvider {
  private final T payload;
  public PayloadApplicationEvent(Object source, T payload) {
    super(source);
    Assert.notNull(payload, "Payload must not be null");
    this.payload = payload;
  }
  @Override public ResolvableType getResolvableType() {
    return ResolvableType.forClassWithGenerics(
        getClass(), ResolvableType.forInstance(getPayload()));
  }
  public T getPayload() {
    return this.payload;
  }
}
```

Listing 3.10 https://github.com/spring-projects/spring-framework/blob/main/spring-context/src/main/java/org/springframework/context/PayloadApplicationEvent.java

Class `PayloadApplicationEvent` is derived from `ApplicationEvent`; if we write our own data types for events, we don't necessarily need to do that. The generic event classes must provide enough type information to the Spring Framework because Java implements generics by type deletion. To provide as much type information as possible, these generic event classes implement the `ResolvableTypeProvider` interface.

---

[32] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/context/PayloadApplicationEvent.html*

The constructor has two parameters: the source of the event and the actual payload of type `T`. The constructor passes the event source to the `ApplicationEvent` superclass, which stores it internally. The next check guarantees that the payload isn't `null`, and the payload is stored in the internal final variable `payload`. The payload can be retrieved via `getPayload()`. It's important for the generic event classes that the `getResolvableType()` method is implemented by the `ResolvableTypeProvider`. The implementation obtains the `Class` object at runtime and also the data type of the `payload` and creates a `ResolvableType` from it.

Using it is a relatively simple process. Here are two listeners that can respond to these events:

```
@Component class ListenerOne {
  @EventListener
  public void onEvent( PayloadApplicationEvent<String> event ) {
    var payload = event.getPayload(); …
  }
}
@Component class ListenerTwo {
  @EventListener
  public void onEvent( PayloadApplicationEvent<UUID> event ) { … }
}
```

The listeners listen to the `PayloadApplicationEvent` once for a payload of type `String` and once for a payload of type `UUID`. In the body of the method, we get to the dispatched event with `getPayload()`.

When we want to send the events, we still use `ApplicationEventPublisher`. Here's an example:

```
PayloadApplicationEvent<String> event1 = new PayloadApplicationEvent<>(
    this, "Clippit" );
publisher.publishEvent( event1 );

PayloadApplicationEvent<UUID> event2 = new PayloadApplicationEvent<>(
    this, UUID.randomUUID() );
publisher.publishEvent( event2 );
```

If we want to send the event, we build one with `new PayloadApplicationEvent`. If the source for the event is our own object, `this` is passed. As the second argument, we pass a string for the first event and a UUID for the second. `publishEvent(…)` will send the event. Spring will ask for the type of the passed object, do a resolution, and then call the correct listener.

3.4.9 Event Transformations

Spring's event processing isn't very compelling, yet there are some interesting possibilities. One of them is the transformation of events. Chains are desirable so that an event is sent on one side, which is accepted and processed by someone else, and the receiver generates a new event. At best, this is declarative, but the Spring Framework doesn't have this capability at its core. What Spring can do, however, is generate a new event at the end of a listener, which is then redistributed by the framework. Schematically, it looks like this:

```
@EventListener
public EventB onContextEvent( EventA event ) {
  …
  return new EventB( … );
}
```

The methods annotated with `@EventListener` can do the following:

- Be declared `void`.
- Return an event object.
- Provide a collection or array of event objects.

Our example uses the second option: return an event object. A collection or array of event objects is also possible as a return.

This allows transformations, as method `onContextEvent(…)` with `@EventListener` shows. This method receives an event, here symbolically `EventA`. This `EventA` is processed, and at the end, something of type `EventB` is created. This event object is returned. With just such methods, the Spring Framework will take `EventB` and put it on the bus by itself and then distribute it further to the respective interested parties.

3.4.10 Sequences by @Order

Spring considers the priority at `@Order` for a defined order. This can look like this:

```
@EventListener( MyEvent.class )
@Order( 1000 )
void myListener() {
  …
}
```

This way, you can control which `@EventListener` is called in which order.

3.4.11 Filter Events by Conditions

The events we've sent so far are delivered to all listening handlers. Everything is typed. That is, a listener only receives events that belong to the corresponding type. But you might want to implement other filter criteria, for example, that the event isn't delivered if certain conditions don't hold. This is precisely the purpose of SpEL, which we've already discussed in Chapter 2, Section 2.11:

Let's consider an example:

```
@EventListener( condition = "#event.isUpdated" )
void onNewPhotoEvent( NewPhotoEvent event ) {
  …
}
```

This means a callback to the listener is only intended if property `isUpdated` is set for the triggered event. As a test criterion, we could use variables in SpEL, read properties, or even call methods. For `EventListener`, this condition is written via annotation attribute `condition`; and `condition` contains the SpEL expression.

A SpEL expression can refer to the event object to decide. There are different notations and options:

- In the example, the SpEL expression contains a fallback to the parameter name, here `event`. Alternatively, you can use `#root.args[<index>]`, `args[<index>]`, `#a<index>`, and `#p<index>` (if the parameter name isn't compiled into the bytecode).
- All arguments are accessible via `#root.args` or `args`.
- `#root.event` or `event` can be used for short references to the Spring Framework `ApplicationEvent`, even if an arbitrary `Object` event was dispatched.

The listener is called if the condition is "truthy." In this context, this means that the expression is either `true`, returns the string `on` or `yes`, or returns the value `1`. Incidentally, the same applies if the condition is initialized with an empty string: the listener method is called.

3.4.12 Synchronous and Asynchronous Events

In event handling, Spring calls the callback methods one after the other, that is, synchronously. This means that at the point where `publishEvent(…)` fires and sends an event, the sender is blocked. First, all listeners are notified, and only then `publishEvent(…)` returns. This isn't without problems because if the listeners take a long time, the other listeners are also considered later.

Spring can also implement the processing of the listeners asynchronously via a background thread. This can be written declaratively simply with `@Async` like this:

```
@Async
@EventListener
void onNewPhotoEvent( NewPhotoEvent event ) {
  …
}
```

For Spring to take the `@Async` methods into account, a `@Configuration` must be annotated with `@EnableAsync`. We'll come back to this in Chapter 4, Section 4.3.1.

There are two things to consider:

- If you use asynchronous methods, you can't use an event transformation. It's not that restrictive because you can still inject an `ApplicationEventPublisher` and publish an event at the end of the asynchronous method itself.
- If an exception occurs in the asynchronous callback method, then it's swallowed and not reported. However, this is a strategy that can be changed via an `AsyncUncaughtExceptionHandler`. We'll return to this data type later in Chapter 4, Section 4.4.2.

3.4.13 ApplicationEventMulticaster *

If you look behind the scenes, an implementation has to take care of managing the listeners and sending the events. In the case of Spring, it's data type `ApplicationEventMulticaster`;[33] it manages the listeners and dispatches events. This type is an interface that is located directly in the Spring Framework core package at `org.springframework.context.event`. Currently, there is only one implementation of this interface: `SimpleApplicationEventMulticaster` (an abstract class is still in between, but we can ignore that).

This default `SimpleApplicationEventMulticaster` can be customized and exchanged, which is useful when you want to have all event handlers automatically called asynchronously. If you want to have asynchronous processing, you always have to annotate the methods with `@Async`; this would no longer be necessary with a customized `SimpleApplicationEventMulticaster.`

This is what it could look like:

```
@Bean
ApplicationEventMulticaster applicationEventMulticaster() {
  var multicaster = new SimpleApplicationEventMulticaster();
  multicaster.setTaskExecutor( new SimpleAsyncTaskExecutor() );
  multicaster.setErrorHandler(TaskUtils.LOG_AND_PROPAGATE_ERROR_HANDLER);
  return multicaster;
}
```

We can build our own bean called `applicationEventMulticaster`. This name is important because it's not the data type that is relevant, but the name of the bean. In the next step, we build an instance of `SimpleApplicationEventMultiCaster` and set a `TaskExecutor` with `setTaskExecutor(…)`; for concurrent execution, we set a `SimpleAsyncTaskExecutor`. You can find more details about the `TaskExecutor` type in Chapter 4, Section 4.4.

---

[33] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/context/event/ApplicationEventMulticaster.html*

The next setting is optional. If something goes wrong with the invoked callback methods, does this information from this thread then get to another thread, or is the error reported? `setErrorHandler( TaskUtils.LOG_AND_PROPAGATE_ERROR_HANDLER )` configures that errors are propagated—that is, forwarded—so that our code is aware that there was an error. The abstract class `TaskUtils`[34] has two constants: `LOG_AND_PROPAGATE_ERROR_HANDLER` and `LOG_AND_SUPPRESS_ERROR_HANDLER`.

3.4.14 Send Events via ApplicationContext

`ApplicationEventPublisher` can be injected into a component at any time. But there is a second way to send events, which is via the `ApplicationContext`. `ApplicationContext` also extends the `ApplicationEventPublisher` interface. So, an alternative way to send events looks like this:

```
ApplicationContext ctx = SpringApplication.run( … );
ctx.publishEvent( myEvent );
```

3.4.15 Attach Listener to SpringApplication *

We can dispatch events through the context and also register listeners with the `SpringApplication` object. `SpringApplication` has three methods that deal with listeners:

- `void setListeners(Collection<? extends ApplicationListener<?>> listeners)`
- `void addListeners(ApplicationListener<?>... listeners)`
- `Set<ApplicationListener<?>> getListeners()`

Here's a fictional example:

```
SpringApplication app = new SpringApplication( … );
ApplicationListener listener = event -> …
app.addListeners( listener );
app.run( args );
```

A similar method for adding listeners also exists in `SpringApplicationBuilder` where the method is simply called `listeners(…)`. The question is, what benefit should `SpringApplication` provide?

org.springframework.boot.context.event

Spring Boot resolves events, in addition to the events that come from Spring Framework. For example, these include events such as `ApplicationStartingEvent`, `ApplicationEnvi-`

---

[34] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/scheduling/support/TaskUtils.html*

`ronmentPreparedEvent`, and so on. The reference documentation[35] lists these types and explains the order. Spring Boot events are located in the `org.springframework.boot.context.event`[36] package. There are other types, specifically in the web environment, but we can ignore those. The special events are subclasses of `SpringApplicationEvent`, which itself is a subclass of `ApplicationEvent`.

Listening for these special events is rarely necessary. However, if it's sometimes necessary, we should consider that some of these events are still sent out before the application context was started up, and thus before the appropriate components were announced and recognized. Then, a `@Component` or `@Bean` as `@EventListener` doesn't work because these components don't exist in this early stage yet. In this case, a login via the `SpringApplication` is necessary.

3.4.16 Listener in spring.factories *

For signing up the listeners, we learned about two different techniques: we can be declarative about the annotations, and we can use a `SpringApplication` object to set the listeners. There is a third quite practical way, which is declarative and doesn't require any code at all: a subdirectory *META-INF* can be created in the classpath (*src/main/resources*, later *target/classes*), in which a *spring.factories* file is located. In this file, behind key `org.springframework.context.ApplicationListener`, a comma-separated list with fully qualified class names of the listeners is written. This is what the file content may look like:

```
org.springframework.context.ApplicationListener=\
 a.b.c.d.e.MyListener
```

The *spring.factories* file is automatically evaluated when the container starts and the listeners are automatically called.

Listeners of Spring Boot

Spring Boot itself also uses this mechanism, with entries in a *spring. factories* file. This listing shows the file snippet from dependency `org.springframework.boot:spring-boot`.

In Listing 3.11, you can see that this key is preconfigured with a whole set of special listeners, and these are then listeners that take over various internal tasks.

```
# Application Listeners
org.springframework.context.ApplicationListener=\
org.springframework.boot.ClearCachesApplicationListener,\
org.springframework.boot.builder.ParentContextCloserApplicationListener,\
org.springframework.boot.context.FileEncodingApplicationListener,\
```

---

[35] *https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#features.spring-application.application-events-and-listeners*

[36] *https://docs.spring.io/spring-boot/docs/current/api/org/springframework/boot/context/event/package-summary.html*

```
org.springframework.boot.context.config.AnsiOutputApplicationListener,\
org.springframework.boot.context.config.DelegatingApplicationListener,\
org.springframework.boot.context.logging.LoggingApplicationListener,\
org.springframework.boot.env.EnvironmentPostProcessorApplicationListener
```

Listing 3.11 https://github.com/spring-projects/spring-boot/blob/main/spring-boot-project/spring-boot/src/main/resources/META-INF/spring.factories

3.5 Resource Abstraction with Resource

In this section, we'll look at the Spring Framework type `Resource`. The type helps to abstract from concrete input/output resources. Looking at the Java standard library, an `InputStream` is a general type for read access to the bytes of a resource. The `InputStream` can be used to read the data using the `read(…)` methods.

Java SE can provide `InputStreams` for a whole range of resources. However, the API for this is always a bit different:

- If a `Path` object exists from a file system, `Files.newInputStream(…)` opens an `InputStream` from this path, and the file contents can be read.
- With a `URL` object, method `openStream(…)` can be used, and the result is also an `InputStream`.
- If there is a byte array, and we want to process this byte array as an `InputStream`, then the class `ByteArrayInputStream` helps; the subclass of `InputStream` also provides `read(…)` methods.
- If resources are in the classpath, then we can work either via a `Class` object or via a `ClassLoader` object with the `getResourceAsStream(…)` method. The method is passed a file name, and the result is an `InputStream` or `null` if the file isn't present.

While there is an `InputStream` at the end in all cases, the API to open it always looks a little different. The problem is that the `InputStream` is already an opened stream. What the Java library lacks is a kind of factory for `InputStreams` that provides streams, but doesn't open them yet. This is exactly what the Spring Framework generalizes with the `Resource` data type.

3.5.1 InputStreamSource and Resource

Spring Framework provides two important data types in the `org.springframework.core.io package`: `InputStreamSource`[37] and `Resource`[38] (see Figure 3.11). `InputStreamSource` can be thought of as a `Supplier`, that is, a contributor of `InputStream` instances.

---

[37] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/core/io/InputStreamSource.html*

[38] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/core/io/Resource.html*

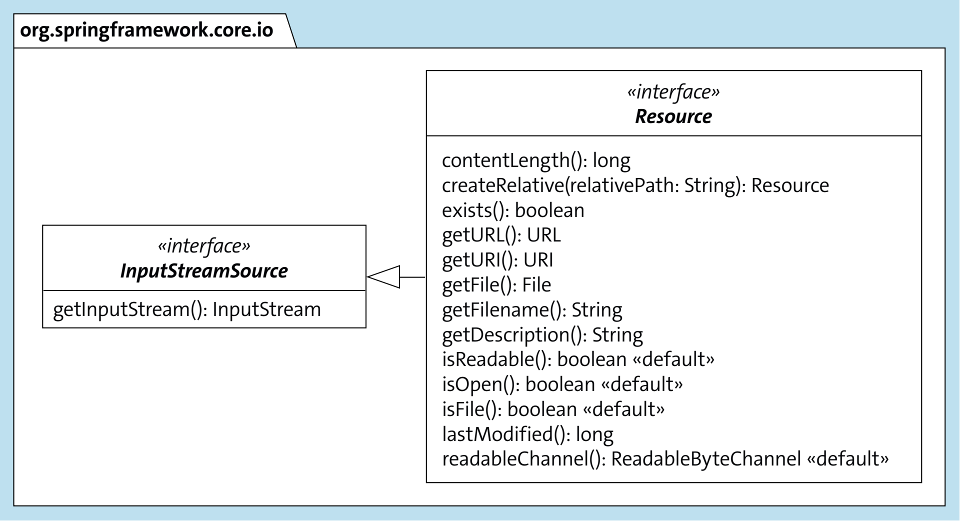

org.springframework.core.io
«interface»
InputStreamSource
getInputStream(): InputStream
«interface»
Resource
contentLength(): long
createRelative(relativePath: String): Resource
exists(): boolean
getURL(): URL
getURI(): URI
getFile(): File
getFilename(): String
getDescription(): String
isReadable(): boolean «default»
isOpen(): boolean «default»
isFile(): boolean «default»
lastModified(): long
readableChannel(): ReadableByteChannel «default»

Figure 3.11 “Resource” and “InputStreamSource”

An `InputStreamSource` has a `getInputStream()` method that opens the data stream. The `Resource` data type extends the `InputStreamSource` interface, and it adds various methods that can do much more than simply open an `InputStream`. To gain a better understanding of the available concrete `Resource` data types, see Figure 3.12.

At the top is `InputStreamSource` with `Resource` directly below. From `Resource`, there is a data type for writable resources: `WriteableResource`. From interface `Resource`, there is also an abstract base class for further subclasses—these are more interesting for us:

- `FileSystemResource` (uses `java.io.File` and `java.nio.file.Path`) and `PathResource` (uses `java.nio.file.Path`) can read from and write to the file system.
- The `ByteArrayResource` can be used to read from byte arrays, comparable to the `ByteArrayInputStream` class.
- `ClassPathResource` reads resources from the classpath, and `UrlResource` gets the data behind a URL.

These data types can be instantiated with `new`, and then we would have a `Resource` instance that we can use. The data stream isn’t yet opened, but this can be done by calling method `getInputStream(…)`. In other words, `Resource` is an abstraction of different resources, and we have a uniform API to open this resource.

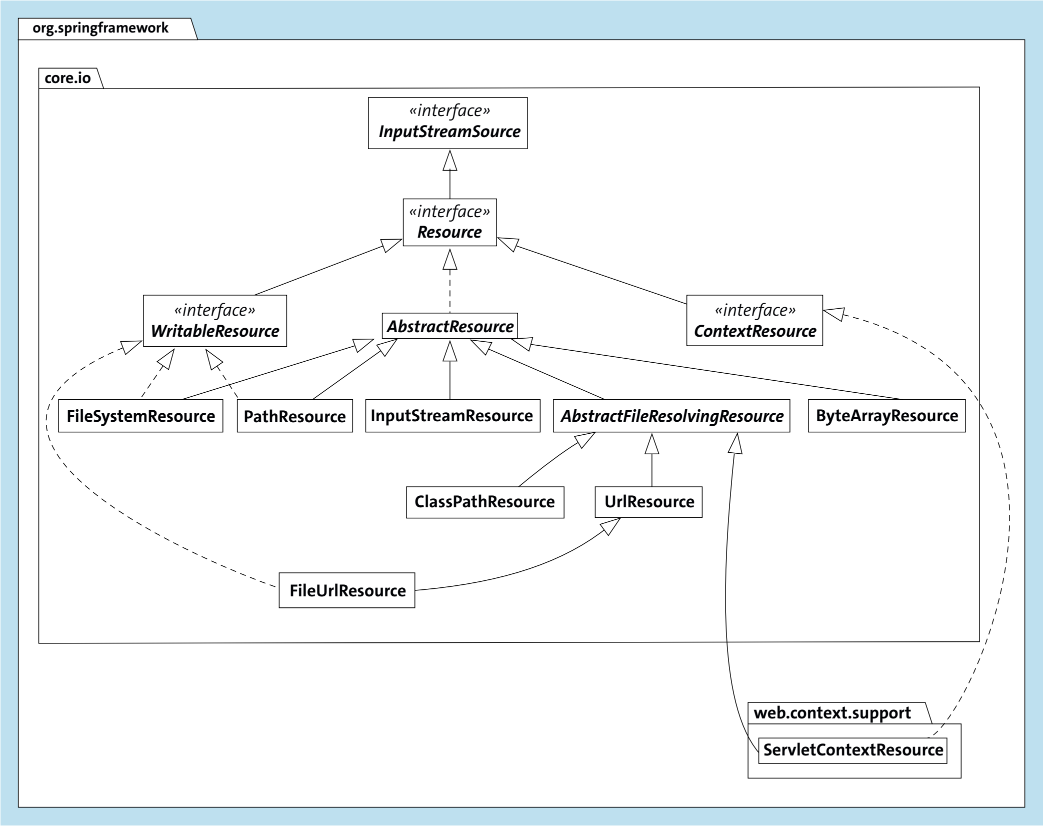

org.springframework
core.io
«interface»
InputStreamSource
«interface»
Resource
«interface»
WritableResource
AbstractResource
«interface»
ContextResource
FileSystemResource
PathResource
InputStreamResource
AbstractFileResolvingResource
ByteArrayResource
ClassPathResource
UrlResource
FileUrlResource
web.context.support
ServletContextResource

Figure 3.12 Implementations of "Resource"

3.5.2 Load Resources

There are three different ways to get a `Resource` object and then ultimately open the input stream with `getInputStream()`. The first option is that we build an instance directly with `new`, for example, for `ClassPathResource`:

```
Resource r = new ClassPathResource( "quotes.csv" );
```

The resource is to be located in the classpath. On the left side of the variable declaration, reference type `Resource` can remain.

The second option is to use the `ResourceLoader` data type:

```
ResourceLoader resourceLoader = new DefaultResourceLoader();
Resource r = resourceLoader.getResource("classpath:quotes.csv");
```

There are two ways to get a `ResourceLoader`. Once you can instantiate an implementation such as `DefaultResourceLoader` as just shown, or you can wire a `ResourceLoader` like this:

```
@Autowired ResourceLoader resourceLoader
```

A `ResourceLoader` declares the `getResource(…)` method and uses it to obtain a `Resource`. The passed string contains information regarding where this resource is obtained from, for example, from the classpath.

The third option is that we forget about these implementing `Resource` classes and the `ResourceLoader` completely, and let `@Value` inject a `Resource`.

3.5.3 Inject Resources via @Value

The following four variable declarations show how `@Value` can be used for the `Resource` data type—of course, the other injection types would have worked instead of field injection:

```
@Value( "classpath:folder/file.txt" )
private Resource r1;

@Value( "file:c:/folder/file.txt" )
private Resource r2;

@Value( "url:http://tutego.com/index.html" )
private Resource r3;

@Value( "classpath*:folder/*.txt" )    // ! * !
private Resource[] r4;
```

For `@Value`, the source is specified, and Spring automatically provides us with the appropriate `Resource` implementation based on this specification.

The first case is a resource from the classpath, and the second specification is a resource from the local file system. In the case of the URL prefix `file`, the local file is loaded via the `UrlResource` class. In the third case, the resource is from the internet.

The Spring Framework uses these prefixes to recognize which concrete `Resource` implementation is needed, instantiates the object, and uses it to initialize the `Resource` variable.

The last example is particularly interesting because it automatically obtains a collection of resources. From the file system search, something like this is familiar: `*.txt`, for example, denotes “all text files in a directory.” The notation is also called *glob syntax*.[39] Using glob syntax in Spring can be particularly useful when dealing with a large number of resources. Instead of manually specifying each resource, we can use a pattern that matches a set of resources based on a specific criterion. This makes it easier to manage and maintain the code, especially when dealing with complex applications that require many resources. The asterisk next to `classpath*` is important because then Spring fills an array of resources with the text files in the directory.

---

[39] *https://en.wikipedia.org/wiki/Glob_(programming)*

The benefit of `Resource` is its abstraction, that is, anything of type `Resource` can be opened with `getInputStream()`. Where the resource is coming from is irrelevant. It can be static or come from a property variable.

Ant Style Patterns

We've just seen that we can name more than one resource with an asterisk. There is a special syntax for this in Spring called *Ant-Style path pattern*. The name is derived from the former build tool *Ant*, which is now considered obsolete. Ant-style path patterns reference an entire group of files, even in subdirectories.

The Javadoc[40] lists the following examples:

1. **`/WEB-INF/*-context.xml`**
   A prefix such as `classpath:` is missing, and then the `ApplicationContext` decides what the root directory is and where this resource is loaded from. The asterisk expresses that everything in the *WEB-INF* directory ending in *-context.xml* is recognized as a resource.
2. **`com/mycompany/**/applicationContext.xml`**
   With two asterisks, the search goes as deep as you want. In this example, all *applicationContext.xml* files are found, no matter how far they are nested under *com/mycompany*.
3. **`file:C:/some/path/*-context.xml`**
   If absolute path specifications are required, the prefix `file` is used. This allows you to target directories in the file system and, for example, find all files in the directory *C:/some/path/* that end with *-context.xml*.
4. **`classpath:com/mycompany/**/applicationContext.xml`**
   Deep searches are also possible in the classpath. Spring proceeds by taking the prefix up to this wildcard and searching recursively from there, depending on the pattern. It's important that recursive traversal of the file tree must be possible in principle. If the resource comes from a classpath of a JAR, a search is possible, as well as in the local file system. These Ant-style path patterns can't be used with a URL to a web page, of course.

3.6 Type Conversion with ConversionService

The `ConversionService` is an essential service in Spring that performs type conversions. It's commonly used in various parts of an application where data needs to be converted from one type to another. For instance, in configuration files, data is stored as a string, and the conversion service can be used to convert this string data to other complex data types, such as integers or `DataSize` objects.

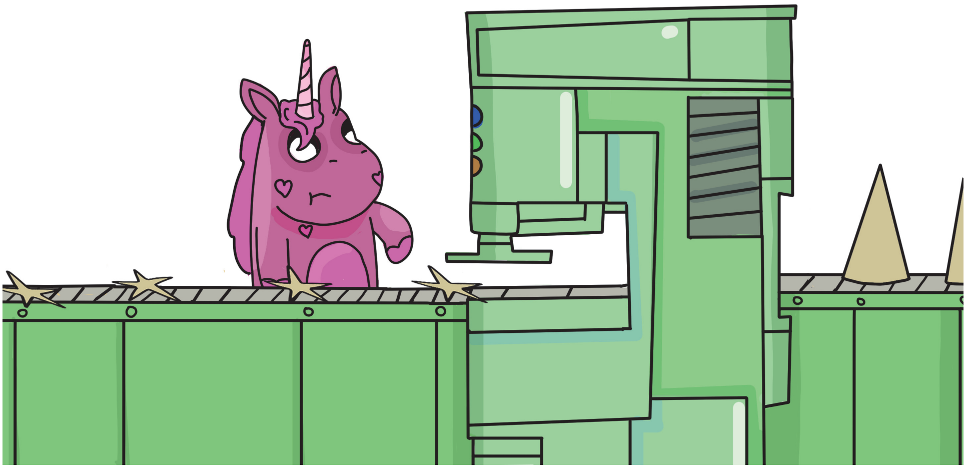
---

[40] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/core/io/support/PathMatchingResourcePatternResolver.html*

The `ConversionService` can also be used to convert string data received from shell methods or web services into other formats. For example, when a shell method is called, a string is passed, which may need to be converted to another format for processing. Similarly, when a web service receives a URL, it's just text, but the conversion service can be used to convert it to a different format.

One of the significant advantages of the `ConversionService` is that it provides a consistent approach to type conversion throughout the application. Instead of manually performing type conversions, developers can rely on the conversion service to handle the conversion process for them.

In Spring, the `ConversionService` can be customized to support custom conversions for specific data types. This allows developers to tailor the conversion process to their specific needs, making the application more efficient and effective.



3.6.1 ConversionService

For type conversions from a type `A` to a type `B`, Spring declares the interface `ConversionService`.[41] This type is part of the Spring Framework and is located in the `org.springframework.core.convert` package. We can observe four methods in the `ConversionService`, as Figure 3.13 shows.

---

[41] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/core/convert/ConversionService.html*

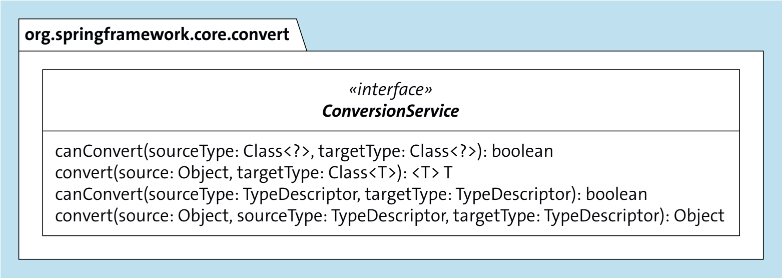

org.springframework.core.convert
«interface»
ConversionService
canConvert(sourceType: Class<?>, targetType: Class<?>): boolean
convert(source: Object, targetType: Class<T>): <T> T
canConvert(sourceType: TypeDescriptor, targetType: TypeDescriptor): boolean
convert(source: Object, sourceType: TypeDescriptor, targetType: TypeDescriptor): Object

Figure 3.13 Methods of the “ConversionService” Interface

Two `boolean` methods are provided to verify if a conversion is possible in principle. In addition, two `convert(…)` methods are available. These methods accept a source and a target representation in the form of a `Class` object or `TypeDescriptor`. Upon conversion, the target type is returned.

3.6.2 DefaultConversionService and ApplicationConversionService

For the implementation of `ConversionService`, the Spring Framework and also Spring Boot offer different implementations. A typical implementation is `DefaultConversionService`;[42] another implementation (specific to Spring Boot) is `ApplicationConversionService`.[43]

The special feature of both implementations is that many converters are already registered: about 50 converters are registered with `DefaultConversionService` and almost three times as many with Spring Boot, namely with `ApplicationConversionService`. There is no method that returns all registered converters, so it’s not easy to find out for which type converters exist. The easiest way is to call the `toString()` method on the `ConversionService` implementation.

Note: Example

We can call the `toString()` representation from `DefaultConversionService` as follows and process the return to get the list of converters:

```
var conversionService = new DefaultConversionService();
conversionService.toString().lines().skip( 1 ).forEach( log::info );
```

The stream expression skips the first line and outputs the converters to the screen. This is how the output begins:

```
java.lang.Boolean -
> java.lang.String : o.s.c.convert.support.ObjectToStringConverter@7d151a
```

---

[42] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/core/convert/support/DefaultConversionService.html*

[43] *https://docs.spring.io/spring-boot/docs/current/api/org/springframework/boot/convert/ApplicationConversionService.html*

```
java.lang.Character -
> java.lang.Number : o.s.c.convert.support.CharacterToNumberFactory@51e37590
java.lang.Character -
> java.lang.String : o.s.c.convert.support.ObjectToStringConverter@40620d8e
```

For example, you can find a converter that can convert a `Boolean` to a `String`, a `Character` to a `Number`, a `Character` to a `String`, and so on.

Sample Conversions

Here are a few examples to illustrate the concept:

```
var converter = new DefaultConversionService();
```

A `DefaultConversionService` instance is used here, but we could also have taken Spring Boot's `ApplicationConversionService`. Now, let's perform the conversion:

```
Properties p = converter.convert( "user=chris", Properties.class );
// {user=chris}
```

After the conversion service has been created, the `convert(…)` method can be called. The first parameter of `convert(…)` is always the object that is to be converted, and the second is a type token that denotes the target type. In this example, the key-value pair `user=chris` is to be transferred to a `Properties` object.

Here are a few more examples:

```
LocalDate ld = converter.convert( LocalDateTime.now(), LocalDate.class );
// 2023-06-05

int[] ia = converter.convert( "1,2,3", int[].class );
// [1, 2, 3]

Currency c = converter.convert( "EUR", Currency.class );
// EUR

Locale l = converter.convert( "GERMANY", Locale.class );
// germany
```

There are a huge number of converters, and the likelihood is high that the Spring Framework or Spring Boot can do conversions on their own.

Conversion with TypeDescriptor

If we look at the target types, we've used `Class` objects so far; however, not every type can be described with them. Generic information, for example, can't be expressed precisely using a `Class` object.

The next example shows that a `Class` object can't express that a list contains strings:

```
List<Integer> input = List.of( 1, 2, 3 );
conversionService.convert( input, List<String>.class );
```

The expression `<String>.class` leads to a compiler error. This is because only a raw type `List.class` is allowed, and no generic details in angle brackets are allowed. Therefore, the `ConversionService` supports a `TypeDescriptor` in addition to the `Class` object for specifying type information. The `TypeDescriptor` object can be used to express `List<String>` like this:

```
List<String> listOfStrings =
  (List<String>) conversionService.convert(
    input,
    TypeDescriptor.forObject( input ),
    TypeDescriptor.collection(
        List.class,
        TypeDescriptor.valueOf( String.class ) ) );
```

The first argument `input` is the source as known. `TypeDescriptor.forObject(input)` specifies the type information of the source. The third argument is the type information for the target type. The specification expresses that a collection of type `List` is desired, which—and for this, a second `TypeDescriptor` is needed—contains strings.

The only downer is that this `convert(…)` method only returns a `java.lang.Object`, and we need an explicit type conversion. But we get the desired object type this way.

3.6.3 ConversionService as Spring-Managed Bean

A `ConversionService` is practical. The only downside is that a `ConversionService` doesn't necessarily exist. That is, if we have a regular Spring Boot application, we can have quite a few things injected, but a `ConversionService` isn't a given.

Spring does a fair amount of conversion internally, and the framework often falls back on property editors, which were mentioned earlier in this chapter. This is an old data type for conversion from the JavaBeans standard. However, if a `ConversionService` is registered, Spring resorts to it because it's significantly more powerful than the preconfigured property editors.

Whether a `ConversionService` was created automatically or not depends a bit on the application. With web applications and with Spring Shell applications, there is automatically a `ConversionService`. These can then also be injected and used for your own programs.

Suppose we've developed a basic Spring Boot application, and its external dependencies don't include a Spring-managed bean for `ConversionService`. In such a scenario, if we want to use a `ConversionService`, we need to create one ourselves and integrate it into the context. This will enable the bean to be injected into other components as required.

The following `@Bean` method returns a `ConversionService`—three variants are shown here:

```
@Configuration
class ConversionServiceConfig {
  @Bean
  @ConditionalOnMissingBean
  public ConversionService conversionService() {
   return
     ApplicationConversionService.getSharedInstance(); // ❶
 // new DefaultConversionService();                     ❷
 // DefaultConversionService.getSharedInstance();       ❸

  }
}
```

For the `@Bean method`, we fall back to `@ConditionalOnMissingBean`, a topic we discussed in Chapter 2, Section 2.10.2. If a `ConversionService` already exist, then the custom `@Bean` method won't be called.

There are different ways to obtain or build a `ConversionService`:

❶ The `ApplicationConversionService` class comes from Spring Boot and declares singleton method `getSharedInstance()`, which returns a `ConversionService`.

❷ Create a new instance of the `DefaultConversionService` type.

❸ Spring itself uses this method internally in some places, and that is to fall back to a global `ConversionService` object from `DefaultConversionService` that returns `getSharedInstance()`. The option should be considered only as a fallback—the first two solutions are better.

3.6.4 Register Your Own Converters with the ConverterRegistry

The `DefaultConversionService` and the `ApplicationConversionService` provide plenty of registered converters. Nevertheless, it may be necessary to register your own converters that can map your own data types, for example.

Including additional converters is a straightforward process that involves invoking an appropriate `addConverter(…)` method. The `ConverterRegistry`[44] type is of primary importance in this context. Figure 3.14 displays the relevant type relationships that we'll examine now.

---

[44] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/core/convert/converter/ConverterRegistry.html*

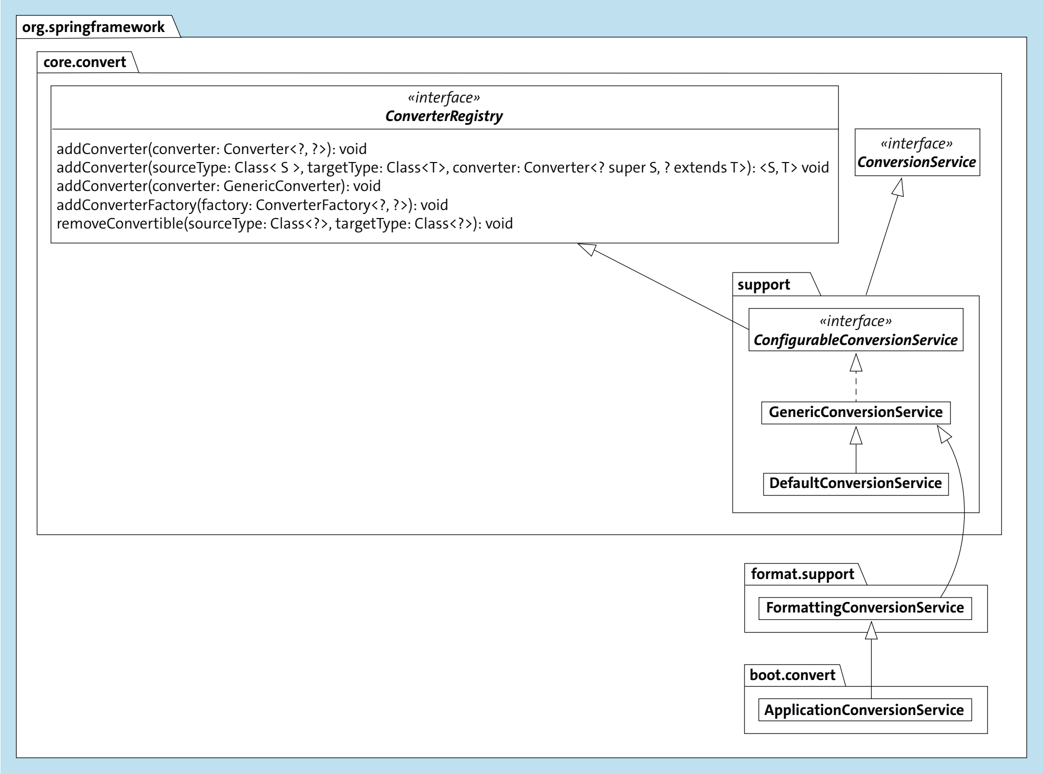

org.springframework
core.convert
«interface»
ConverterRegistry
addConverter(converter: Converter<?, ?>): void
addConverter(sourceType: Class< S >, targetType: Class<T>, converter: Converter<? super S, ? extends T>): <S, T> void
addConverter(converter: GenericConverter): void
addConverterFactory(factory: ConverterFactory<?, ?>): void
removeConvertible(sourceType: Class<?>, targetType: Class<?>): void
«interface»
ConversionService
support
«interface»
ConfigurableConversionService
GenericConversionService
DefaultConversionService
format.support
FormattingConversionService
boot.convert
ApplicationConversionService

Figure 3.14 Type Relations of "ConverterRegistry" and Implementations

`ConverterRegistry` is implemented by the already introduced classes `DefaultConversionService` and `ApplicationConversionService` (package `org.springframework.boot.convert`). For details on these classes, both data types have a common superclass: `GenericConversionService`. This in turn implements the `ConfigurableConversionService` interface; this interface in turn extends two interfaces: `ConversionService` and `ConverterRegistry`. The type `ConversionService` has `convert(…)` methods.

The methods for registering and removing converters are declared in the `ConverterRegistry` interface. It also declares one method for removing a converter and four methods for adding converters. The methods differ in what kind of converters are registered. Three data types can be identified: `Converter`, `GenericConverter`, and `ConverterFactory`. We want to deal with these three data types more intensively.

Converter

The simplest conversion is done using the `Converter`[45] type, which is intended for 1:1 conversions (see Figure 3.15). While a source object comes in as an argument to the methods, the converted object comes out of the method.

---

[45] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/core/convert/converter/Converter.html*

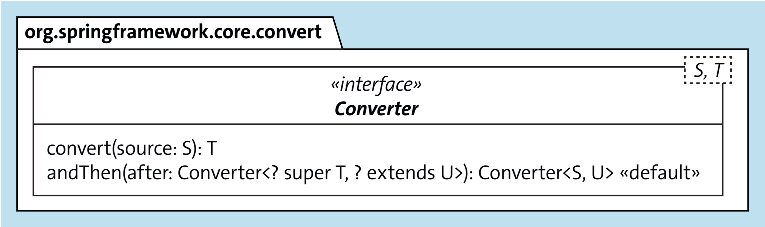

org.springframework.core.convert
S, T
«interface»
Converter
convert(source: S): T
andThen(after: Converter<? super T, ? extends U>): Converter<S, U> «default»

Figure 3.15 “Converter” Interface

The `Converter` type declares two type variables: `S` represents the source and `T` the target.

Here is a basic example from the Spring Framework that demonstrates the implementation of `StringToUUIDConverter`:[46]

```
final class StringToUUIDConverter implements Converter<String, UUID> {

    @Override
    @Nullable
    public UUID convert(String source) {
        return (StringUtils.hasText(source) ?
                        UUID.fromString(source.trim()) : null);
    }

}
```

The class implements the `Converter` interface, and the type arguments make it clear that a `String` should be converted to a `UUID`. The actual `convert(…)` method is implemented from the interface, and the code is rather trivial.

ConverterFactory

The next data type is the `ConverterFactory`.[47] A `ConverterFactory` can be thought of as a 1:n relationship that is used whenever a source type is to be converted into a special type of target hierarchy.

Here are a few examples:

- `String` → to a `Number` subclass
- `Integer` → into an enumeration element of `enum`
- `String` → into an enumeration element of `enum`
- Case-insensitive `String` → into an `enum` element

---

[46] *https://github.com/spring-projects/spring-framework/blob/main/spring-core/src/main/java/org/springframework/core/convert/support/StringToUUIDConverter.java*

[47] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/core/convert/converter/ConverterFactory.html*

Of course, you could write a `Converter` for `String` to `Byte`, `String` to `Short`, and so on, but for a conversion to a `Number` implementation, the `ConverterFactory` is optimal. The `ConverterFactory` is also perfect for enumeration elements: an ordinal number can be mapped to an `enum` object at this position, or the name of the enumeration element can be mapped to an `enum` object.

The `ConverterFactory` interface has a method that returns a `Converter` for a target type:

```
public interface ConverterFactory<S, R> {
  <T extends R> Converter<S, T> getConverter(Class<T> targetType);
}
```

Spring uses this internally, for example, in an `IntegerToEnumConverterFactory`.[48]

GenericConverter

For some conversions, there are many combinations, for example, when an array with elements of type `S` is to be converted to another array with element types `T`, an array to a `Collection`, or a `Collection<S>` to a `Collection<T>`.

There could be an incredible number of combinations, and, for this case, the Spring Framework declares the `GenericConverter` type:

```
public interface GenericConverter {

    @Nullable
    Set<ConvertiblePair> getConvertibleTypes();

    @Nullable
    Object convert(@Nullable Object source,
                   TypeDescriptor sourceType, TypeDescriptor targetType);

    final class ConvertiblePair { … }
}
```

Converter, ConverterFactory, GenericConverter

Implementations of the interfaces `Converter`, `ConverterFactory`, and `GenericConverter` are responsible for the actual conversion. The UML diagram shown in Figure 3.16 summarizes the methods once again.

---

[48] For those interested, see *https://github.com/spring-projects/spring-framework/blob/main/spring-core/src/main/java/org/springframework/core/convert/support/IntegerToEnumConverterFactory.java*.

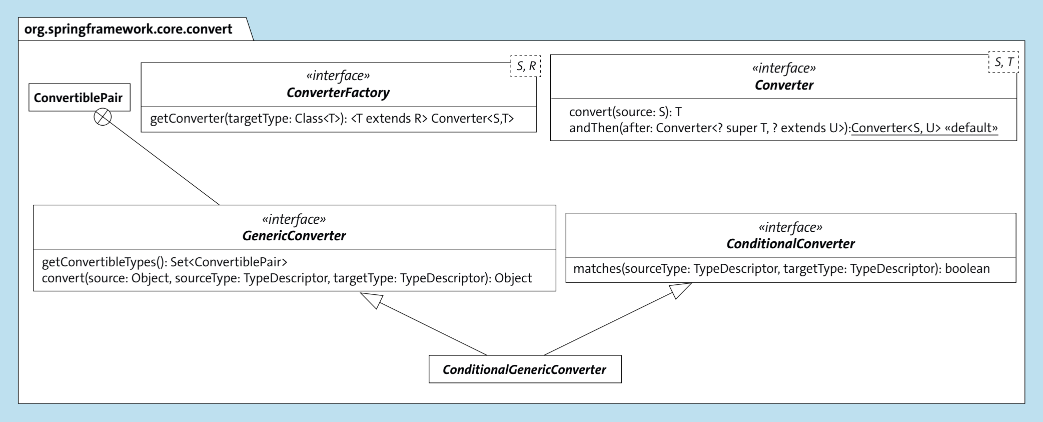

org.springframework.core.convert
ConvertiblePair
S, R
«interface»
ConverterFactory
getConverter(targetType: Class<T>): <T extends R> Converter<S,T>
S, T
«interface»
Converter
convert(source: S): T
andThen(after: Converter<? super T, ? extends U>):Converter<S, U> «default»
«interface»
GenericConverter
getConvertibleTypes(): Set<ConvertiblePair>
convert(source: Object, sourceType: TypeDescriptor, targetType: TypeDescriptor): Object
«interface»
ConditionalConverter
matches(sourceType: TypeDescriptor, targetType: TypeDescriptor): boolean
ConditionalGenericConverter

Figure 3.16 Converter Types

In principle, `GenericConverter` can convert numerous pairs. However, can it actually do this? There may be combinations that don't work at all. The `GenericConverter` can't tell whether it can perform a conversion. That's why there is another data type in the Spring Framework: `ConditionalConverter`. This one can determine with a new method called `matches(…)` if two type descriptions (with a `TypeDescriptor` for the source type and a `TypeDescriptor` for the target type) can be converted at all. A new unification interface `ConditionalGenericConvert` combines the method `matches(…)` and the methods `getConvertableTypes()` and `convert(…)`.

Now that we know how converters work, let's use that for the Spring Shell.

Register Converter in the Spring Shell

We've written different commands in `FsCommands`. The methods for the commands have parameters, and these don't necessarily have to be of type string, but other data types are allowed. This is because the Spring Shell uses a `ConversionService` internally, which converts a string from the command line into, for example, an `int`, `long`, `duration`, and so on.

We want to add a small new method to find out if a path exists:

```
@ShellMethod( "Display if a path exists" )
public String exists( Path path ) {
  boolean exists = …;
  return String.format( "Path to '%s' %s exist",
                        path, exists ? "does" : "doesn't" );
}
```

If you start the application with the preceding code and call the `exists` command, it gives an error message saying that no converter was found that can convert the `java.lang.String` data type to `java.nio.file.Path`.

Conveniently, the Spring Shell automatically registers all components of type `Converter`, `GenericConverter`, or `ConverterFactory` with the `ConversionService`. That means we don't need to inject a `ConverterRegistry` and add a converter with an `add*(…)` method. This is how it can look:

```
@Component
class StringToPathConverter implements Converter<String, Path> {
  @Override public Path convert( String source ) {
    return Path.of( source );
  }
}
```

The converter is a Spring-managed bean and is detected at application startup. The Spring Shell fetches all types that implement `Converter` and automatically registers them with the `ConversionService`. If you call `exists` again, there will be no error.

3.6.5 Printer and Parser

`ConversionService` is a powerful mechanism for converting types, and we could see by example how the Spring Shell uses `ConversionService`. `ConversionService` also appears in other places, such as web requests, reading and parsing configuration data, and SpEL.

When converting from a string to a target format—and when converting from an object to a string—the current language (represented in Java via `Locale`) must also be considered, especially for graphical interfaces. However, `ConversionService` doesn't take this into account because it doesn't work localized.

If the local language is to be considered during conversion, Spring Framework has two additional data types: `Printer` and `Parser`:

- A `Printer` converts an object into a localized `String` representation. A typical example is a floating-point number that uses a decimal comma or decimal point to separate the decimal places, depending on the local language.
- Conversely, a `Parser` converts a `String` with a language back into an object.

The `Printer` and `Parser` data types are fairly simple, as shown in the UML diagram in Figure 3.17.

`Parser` has a `parse(…)` method, a `String` and a `Locale` are passed, and the result is an object of type `T`. Conversely, with `Printer`, an object of type `T` and `Locale` are passed, and a `String` is the result.

Often, `Parser` and `Printer` are used together, and therefore the Spring Framework declares the unification interface `Formatter`.

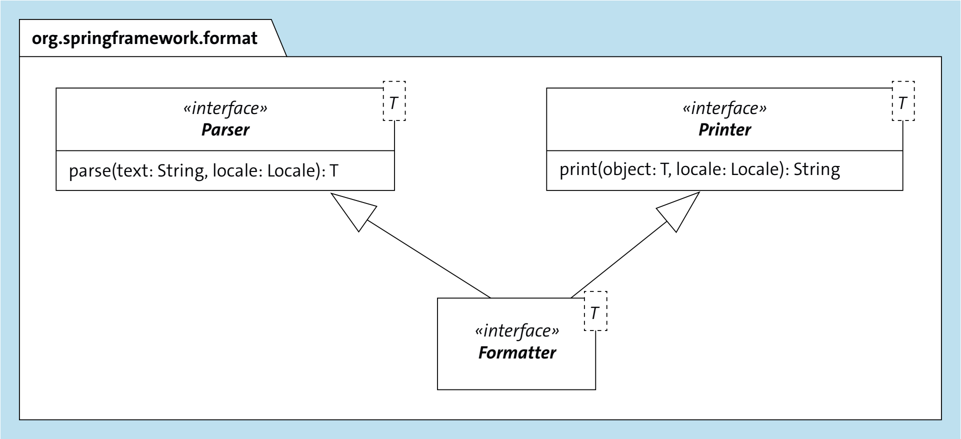

org.springframework.format
«interface»
Parser
T
parse(text: String, locale: Locale): T
«interface»
Printer
T
print(object: T, locale: Locale): String
«interface»
Formatter
T

Figure 3.17 “Formatter”: “Parser” and “Printer”

DurationFormatter

Most `Formatter` implementations are simple to program. The Spring Framework declares a `DurationFormatter` that can convert a `java.time.Duration` to a `String` and a `String` to a `java.time.Duration`. Let’s take a look at the source code:

```
package org.springframework.format.datetime.standard;

import …

/**
 * {@link Formatter} implementation for a JSR-310 {@link Duration},
 * following JSR-310's parsing rules for a Duration.
 *
 * @author Juergen Hoeller
 * @since 4.2.4
 * @see Duration#parse
 */
class DurationFormatter implements Formatter<Duration> {

  @Override
  public Duration parse(String text, Locale locale)
      throws ParseException {
    return Duration.parse(text);
  }

  @Override
  public String print(Duration object, Locale locale) {
    return object.toString();
  }
}
```

https://github.com/spring-projects/spring-framework/blob/main/spring-context/src/main/java/org/springframework/format/datetime/standard/DurationFormatter.java

The code shows that the `Locale` isn't relevant at all. Therefore, the implementation isn't a good example of a `Formatter` that should work localized. But Spring doesn't desire a localization in `Duration` (it's not standardized either), and therefore the implementation can ignore the `Locale`. After an implementation has been made, it must be registered.

3.6.6 FormatterRegistry

Thus far, we've gained knowledge about several data types that are present in the `org.springframework.core.convert` package. To further enhance our understanding, we'll explore the data types that exist within the `org.springframework.format` package. Refer to Figure 3.18 for a comprehensive overview.

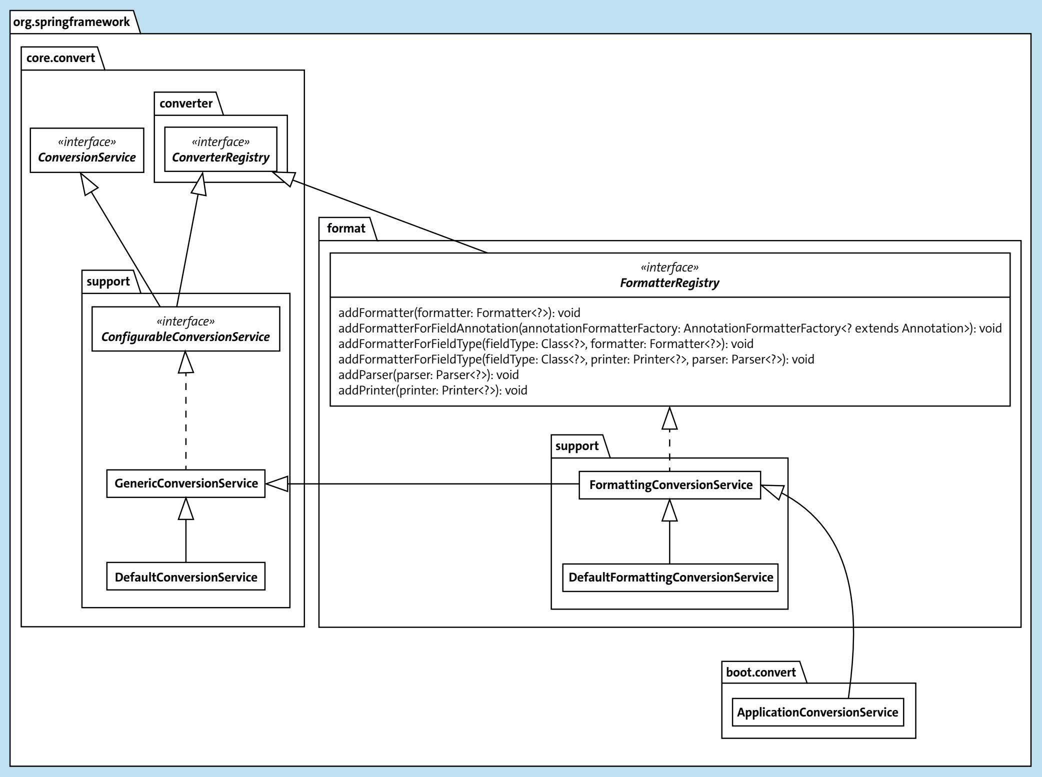

org.springframework
core.convert
converter
«interface»
ConversionService
«interface»
ConverterRegistry
support
«interface»
ConfigurableConversionService
GenericConversionService
DefaultConversionService
format
«interface»
FormatterRegistry
addFormatter(formatter: Formatter<?>): void
addFormatterForFieldAnnotation(annotationFormatterFactory: AnnotationFormatterFactory<? extends Annotation>): void
addFormatterForFieldType(fieldType: Class<?>, formatter: Formatter<?>): void
addFormatterForFieldType(fieldType: Class<?>, printer: Printer<?>, parser: Parser<?>): void
addParser(parser: Parser<?>): void
addPrinter(printer: Printer<?>): void
support
FormattingConversionService
DefaultFormattingConversionService
boot.convert
ApplicationConversionService

Figure 3.18 Converter Types and *Registry Classes

Converters can be registered with `ConverterRegistry`, but not `Formatter`. This is allowed by the `FormatterRegistry` subtype.[49] It has several `addFormatter*(…)` methods for adding `Formatter` instances; In addition, `Printer` and `Parser` can be registered separately.

`FormatterRegistry` is implemented by, among others, `FormattingConversionService`.[50] `FormattingConversionService` is used as base type for `DefaultFormattingConversionService`[51] and by `ApplicationConversionService` of Spring Boot.

`ApplicationConversionService` is more powerful than `DefaultConversionService` because no `Formatter` can be registered with `DefaultConversionService`; the type doesn't implement `FormatterRegistry`.

Use Parser/Printer via ConversionService

`ConversionService` doesn't provide direct access to the converter instances and doesn't share converter instances with the outside world. The usage pattern is different: the `convert(…)` method of `ConversionService` searches for the corresponding converter from an internal data structure and calls `convert(…)` on the determined converter. It's the same with types `Parser` and `Printer`: you can't request `Parser` or `Printer` for a source type and `Locale`. Internally, `Parser` and `Printer` are converted into a `ParserConverter` or `PrinterConverter`, and this is added to the internal data structure. If the `convert(…)` method is called later by `ConversionService`, then `ParserConverter`/`PrinterConverter` are also taken into account.

To summarize, the `ParserConverters` are special `Converter` instances that convert from a `String` (source type) to an object (target type), and the `PrinterConverter` instances convert from an object to a `String` considering a local language.

If `ConversionService` selects `ParserConverter` or `PrinterConverter` for conversion, the current national language is determined and passed to `Parser` or `Printer`. This can be seen in the following example:

```
var converter = ApplicationConversionService.getSharedInstance();
LocaleContextHolder.setLocale( Locale.GERMANY ); // 16.06.24
System.out.println( converter.convert( LocalDate.now(), String.class ) );
LocaleContextHolder.setLocale( Locale.FRANCE ); // 16/06/2024
System.out.println( converter.convert( LocalDate.now(), String.class ) );
```

Because we're working with Spring Boot, we can use `ApplicationConversionService`. Then `LocaleContextHolder.setLocale(…)` sets the current language as a thread-local object.[52]

---

[49] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/format/FormatterRegistry.html*

[50] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/format/FormatterRegistry.html*

[51] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/format/support/DefaultFormattingConversionService.html*

[52] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/context/i18n/LocaleContextHolder.html*

If the language was set, `convert(…)` shows a different formatting for the `LocaleDate` object. This is because `PrintConverter` gets the language again from `LocaleContextHolder`. A code snippet from `FormattingConversionService`[53] illustrates this:

```
private static class PrinterConverter implements GenericConverter {

  …
  @Override
  @SuppressWarnings("unchecked")
  public Object convert( @Nullable Object source, TypeDescriptor sourceType,
                         TypeDescriptor targetType) {
    if (!sourceType.isAssignableTo(this.printerObjectType)) {
      source = this.conversionService.convert(source, sourceType,
                                              this.printerObjectType);
    }
    if (source == null) {
      return "";
    }
    return this.printer.print( source, LocaleContextHolder.getLocale());
  }
  …
}
```

The use of types `Parser` and `Printer` occurs primarily in web applications. We'll come back to this later in Chapter 9, Section 9.9.4.

3.6.7 DataBinder

Finally, we want to come to another data type, namely `DataBinder`,[54] which isn't located under `org.springframework.core.convert` or `org.springframework.format`, but `org.springframework.validation`.

`DataBinder` uses the `ConversionService` internally to fill a JavaBean automatically with values. For `DataBinder`, there is also a subclass `WebDataBinder`, and from this class, there are even more subclasses, so that we can later transfer the request information from a web service directly to a JavaBean.

Suppose we have a `Profile` class that contains an ID and a nickname, along with their respective setter and getter methods:

---

[53] *https://github.com/spring-projects/spring-framework/blob/main/spring-context/src/main/java/org/springframework/format/support/FormattingConversionService.java*

[54] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/validation/DataBinder.html*

```
class Profile {
  long   id;
  String nickname;
  // Setter + Getter
}
```

It's possible to map data pairs such as `id = 1` and `nickname = "Fillmore"` automatically to the profile. The property values are stored in a special hash map. For this purpose, class `MutablePropertyValues`[55] can be used:

```
var propertyValues = new MutablePropertyValues();
propertyValues.add( "id", 1 );
propertyValues.add( "nickname", "Fillmore" );
```

The `MutablePropertyValues` data type accepts key-value pairs with the `add(…)` method; in our case, the ID should be `1`, and the nickname should be `"Fillmore"`.

This `MutablePropertyValues` object has nothing to do with profiles, but only represents the data. For example, a program could fill a `MutablePropertyValues` object with values from a URL or from the command line and later transfer them to a `Profile` object.

Let's proceed with the following steps. First, we'll create a `Profile` instance without any data:

```
var fillmore = new Profile();
```

In a realistic scenario, Java reflection would build the target object.

The `Profile` object can be automatically filled by the `DataBinder` with the data from the `PropertyValues` container:

```
var dataBinder = new DataBinder( fillmore );
dataBinder.setConversionService( new DefaultConversionService() );
// otherwise with PropertyEditor instances
dataBinder.bind( propertyValues );
log.info( "{} has id {}", fillmore.nickname, fillmore.id );
// Fillmore has id 1
```

With `DataBinder`, the container is passed to the constructor. Then, we set `ConversionService`. This isn't mandatory, but we've noticed that converter instances can be registered and perform arbitrary conversions. The example uses `DefaultConversionService`; Spring Boot's `ApplicationConversionService` is of course also possible. If you don't set a `ConversionService`, `DataBinder` internally uses the weaker property editors, and they don't offer as many conversion options.

---

[55] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/beans/MutablePropertyValues.html*

The actual transfer of the values from the `PropertyValues` container to the profile takes place via the `bind(…)` method.

BindingResult in Case of Errors

Assuming that a conversion process fails, it results in an error that is communicated through a specialized container. Consider the following modified example:

```
PropertyValues propertyValues =
    new MutablePropertyValues( Map.of( "id",       "ONE",
                                       "nickname", "Fillmore" ) );
```

For the construction of the `MutablePropertyValues` object, the program uses another way: in the constructor, the key-value pairs, that is, the property assignments, are passed directly as a `Map`. The `id` is incorrect because the value isn't a number, but a `String`, which can't be converted to a `long` in this form. (But if you want to register a converter here, then this would be possible in principle.)

The next lines are identical to the previous program:

```
var fillmore   = new Profile();
var dataBinder = new DataBinder( fillmore );
dataBinder.setConversionService( new DefaultConversionService() );
dataBinder.bind( propertyValues );
```

The program won't work, but the `DataBinder` tries to map as much as possible. "Fillmore" would be bound as a name, but not the ID.

`PropertyValues`[56] can be used with `getPropertyValues()` to ask which `PropertyValue` objects[57] exist; the following log output would show the properties `nickname` and `id`:

```
log.info( "Binding values: {}",
          asList(propertyValues.getPropertyValues()) );

// Binding values: [bean property 'nickname', bean property 'id']
```

If you want to know something about the success of a binding, the `DataBinder` method `getBindingResult(…)` returns a `BindingResult` object as a result. `BindingResult`[58] is an interface with a `getModel(…)` method, and this method returns a simple `java.util.Map` with error information:

```
log.info( "{}", dataBinder.getBindingResult().getModel() );
```

---

[56] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/beans/PropertyValues.html*

[57] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/beans/PropertyValue.html*

[58] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/validation/BindingResult.html*

The output shows the following:

```
// {target=Fillmore has id 0, o.s.v.BindingResult.target=
// o.s.v.BeanPropertyBindingResult: 1 errors
// Field error in object 'target' on field 'id': rejected value [ONE]; codes
// [typeMismatch.target.id,typeMismatch.id,typeMismatch.long,typeMismatch];
// arguments [o.s.context.support.DefaultMessageSourceResolvable:
// codes [target.id,id]; arguments []; default message [id]]; default
// message [Failed to convert property value of type 'java.lang.String'
// to required type 'long' for property 'id'; nested exception is
// java.lang.NumberFormatException: For input string: "ONE"]}
```

In the log, it can be seen that binding the ID with string `ONE` to the `long` property `id` of the object doesn't work. The log output also shows that the `DataBinder` tries `Long.parseLong(…)`, but the result is a `NumberFormatException`.

Summary

The conversion capabilities of the Spring Framework are very handy, and `ConversionService` is used in various places in Spring applications. New converters can be added and are automatically taken into account. This is important because, thanks to converters, you no longer have to perform a type conversion yourself in methods—usually at the very beginning. Instead, the task can be delegated to the Spring Framework. From Section 3.6.4, compare

```
@ShellMethod( "Display if a path exists" )
public String exists( Path path ) {
  …
}
```

with the manual work:

```
@ShellMethod( "Display if a path exists" )
public String exists( String pathAsString ) {
  Path path = Path.of( pathAsString );
  …
}
```

This fits well into the philosophy of the IoC: our own program gives up the control to convert. We no longer have control over the conversion, but we tell Spring: "I want something of such-and-such data type," and then a converter makes sure that the data is also transferred to the desired data type.

3.7 Internationalization *

Given the global distribution of software, it's imperative that it doesn't display just one national language. Instead, software should be designed and programmed from the outset so that it can be easily used in different languages. This process is called *internationalization*. In this process, the software is programmed so that output is basically possible in any language. The translation for a specific language is called *localization*.

3.7.1 Possibilities for Internationalization with Java SE

Java SE already provides extensive functionality that saves us from having to program everything ourselves. A `Locale` object expresses a language in a region, and thus tells the Java SE how to format a date or a floating-point number. Java SE supports a wide range of languages.

For international applications, different outputs must be provided for different languages. For this, Java SE provides class `ResourceBundle`. This class is a kind of associative memory that links a key with a translated text.

Spring reaches out to us even more, as the framework allows us to automatically initialize these `ResourceBundle` objects with loaded translations and provides additional utility methods. One particularly interesting feature of Spring is that translations can be updated at application runtime. This makes it possible to change the translations without having to restart the application.

Properties Files with Translations

The core of internationalization lies in resource files, each of which contains the keys and the translated texts. For example, if we want to check in a shell command whether paths exist, we can output this output in English, a standard language, and German.

To achieve this, we place the resource files in the classpath, that is, the sources under *source/main/resources*, so that they later go into the *target directory*. For better grouping, we create another directory called *messages* and below that a subdirectory named *shell*. The directory names can be freely chosen, but *messages* works well as a root directory for internationalized texts, while *shell* indicates that it's about translated shell commands. The properties file is called *fscommands* because it contains translations related to file system commands. This resource file isn't associated with any specific language and is the *default resource file*:

```
shell.fs.path-exist=Path ''{0}'' does exist
shell.fs.path-not-exist=Path ''{0}'' doesn’t exist
```

Listing 3.12 src/main/resources/messages/shell/fscommands.properties

It’s important to have some hierarchy in the keys to prevent possible key conflicts. Therefore, we use prefix `shell.fs`.

In Spring, the key is called *code* and is associated with the actual message. We can insert placeholders in the message with curly braces, which are later replaced with arguments. To ensure that the substitution works properly, we need to escape single quotes by prefixing them with another quote. This way we get a single quote at the end.

This was the content of the first properties file. There should be a second properties file in the same directory:

```
shell.fs.path-exist=Path ''{0}'' exists
# shell.fs.path-not-exist=path ''{0}'' doesn’t exist
```

Listing 3.13 src/main/resources/messages/shell/fscommands_en_DE.properties

The file name contains a language and country identifier; the suffix `de_DE` stands for German in Germany. There are other countries where German is spoken, and different phrases may be used there.

In the *fscommands_en_DE.properties* file, only the first string will be translated to indicate that the path exists. The second string is intentionally commented out.

Later, Java SE, or Spring, will load both properties files and perform a hierarchical search. For example, if the local language is German, and the key `shell.fs.path-exist` is searched for, the most specific properties file is queried first, and the result is found. However, if `shell.fs.path-not-exist` is searched for, the search mechanism won’t find it in the specific translation file and will search in the following properties file where the corresponding value is found.

Therefore, it’s a best practice to define all keys in the generic language, usually English, and assign a value to them. Then, new files are created for each national language, and the specific keys are overwritten individually. If there is no value for a specific code, this isn’t so bad because it’s always possible to fall back on the default resource file.

3.7.2 MessageSource under Subtypes

Java SE provides the `ResourceBundle` type, but we don’t work with it directly in Spring applications. The Spring Framework abstracts from it and introduces a new data type `MessageSource`.[59] The UML diagram in Figure 3.19 shows the type relationships.

The `getMessage(…)` method of the `MessageSource` base type is overloaded. In one variant, the code, that is, the key, is passed first. Thereafter, the curly brackets in an array are assigned, followed by the specification of the desired language for the translation. A variant allows a default message. The result is a string.

---

[59] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/context/MessageSource.html*

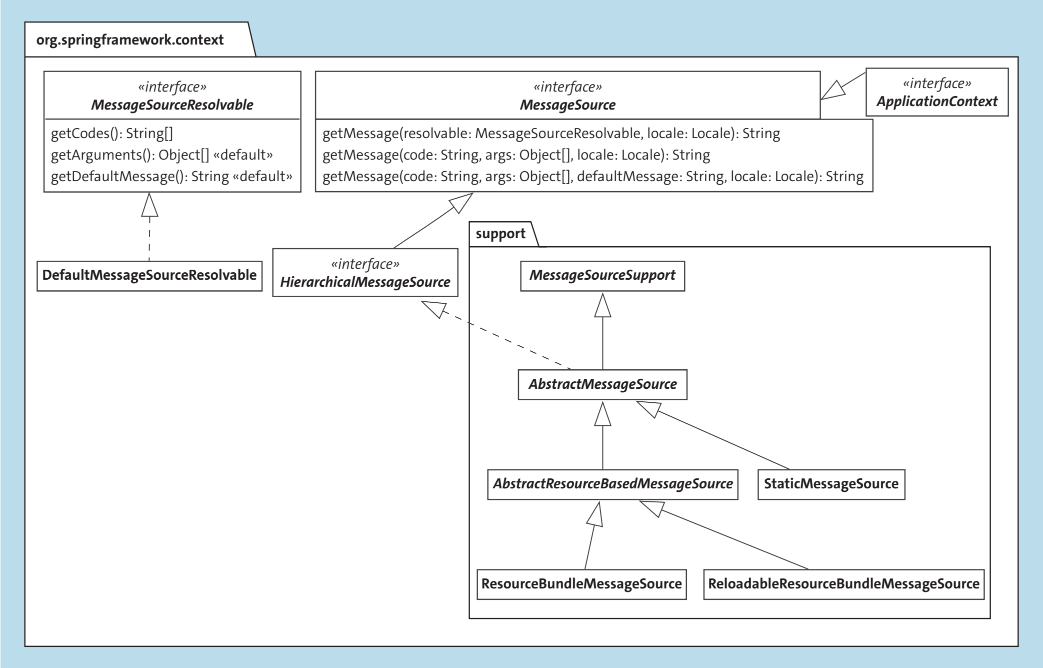

org.springframework.context
«interface»
MessageSourceResolvable
getCodes(): String[]
getArguments(): Object[] «default»
getDefaultMessage(): String «default»
«interface»
MessageSource
getMessage(resolvable: MessageSourceResolvable, locale: Locale): String
getMessage(code: String, args: Object[], locale: Locale): String
getMessage(code: String, args: Object[], defaultMessage: String, locale: Locale): String
«interface»
ApplicationContext
DefaultMessageSourceResolvable
«interface»
HierarchicalMessageSource
support
MessageSourceSupport
AbstractMessageSource
AbstractResourceBasedMessageSource
StaticMessageSource
ResourceBundleMessageSource
ReloadableResourceBundleMessageSource

Figure 3.19 Spring Type “MessageSource” and Subtypes

Another option is to use `MessageSourceResolvable` to encode multiple codes. If there is no translation for the first code, the array can contain a second code that is obtained next.

`MessageSource` has subtypes like `ApplicationContext`, but this one isn’t relevant due to its inflexibility. There are other subtypes, interfaces, and implementations of `MessageSource`, but we’ll focus here on the `ResourceBundleMessageSource` and `ReloadableResourceBundleMessageSource` classes. A third implementation called `StaticMessageSource` is only intended for testing and is rather uninteresting in real operation.

ResourceBundleMessageSource

In the following, we’ll work with the `ResourceBundleMessageSource`[60] class, an implementation of `MessageSource` that is internally based on `ResourceBundle`. An instance can be created using the parameterless constructor:

```
var messageSource = new ResourceBundleMessageSource();
```

In the next step, some parameterization is necessary:

---

[60] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/context/support/ResourceBundleMessageSource.html*

```
messageSource.setBasename( "messages/shell/fscommands" );
messageSource.setDefaultEncoding( "UTF-8" );
messageSource.setUseCodeAsDefaultMessage( true );
```

The *base name*, that is, which translations this `MessageSource` object should represent, is essential. In our example, it's the bundle named `fscommands` in the `messages/shell` directory/package. We don't specify a file name because the mapping to the file name includes the language and that's what the library does; that's why building the file names according to the pattern is so important. The loading order and the structure of the file names can be read from Java SE class `ResourceBundle`.[61]

Besides the ability to specify only one base name for a message bundle, it's also possible to specify several base names separated by commas. In this case, a linear search is performed in the order in which they are specified. Alternatively, you can also search hierarchically by using `setParentMessageSource(…)` and specifying a `MessageSource` as parent.

After we've set the base name, we can also set the encoding of the properties file to UTF-8. By default, the default encoding of the platform is used, which doesn't have to be UTF-8.[62] Lastly, we can set a property with `setUseCodeAsDefaultMessage(…)` to output the key as a string if no translation is available. This doesn't lead to a perfect output, but it helps us in the development, so we can see where a text or a translation is missing.

After the `ResourceBundleMessageSource` object is built and configured, `getMessage(…)` can be used. The language to be passed should come from `LocaleContextHolder` as an example, which in Spring contains the "default" language in a thread-local variable:

```
Locale lang = LocaleContextHolder.getLocale();
String msg1 = messageSource.getMessage( "shell.fs.path-exist",
                                        new Object[]{ "c:/test" }, lang );
String msg2 = messageSource.getMessage( "shell.fs.path-not-exist",
                                        new Object[]{ "c:/test" }, lang );
```

The language is passed as a `Locale` object to the `getMessage(…)` method along with the key and the placeholder assignments.

MessageSource through Spring Boot Auto-configuration

Spring Boot automatically creates a Spring-managed bean of type `MessageSource` via auto-configuration if we haven't registered our own `MessageSource` as a Spring-managed bean. We can therefore wish to have a `MessageSource` instance:

```
@Autowired MessageSource messageSource;
```

---

[61] *https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/ResourceBundle.html#getBundle(java.lang.String,java.util.Locale,java.lang.ClassLoader)*

[62] This changes with new Java versions; see *https://openjdk.org/jeps/400*.

By default, `MessageSource` is of type `ResourceBundleMessageSource`, and base name `messages` is preset, which means that files with format *messages*.properties* are used. We can change the base name by specifying a comma-separated list of base names under key `spring.messages.basename`. For our resource bundle `messages.shell.fscommands`, it can look like this in the *application.properties* file:

```
spring.messages.basename=messages.shell.fscommands
```

After configuration, `MessageSource` can be used as before. An example of this is the following call to retrieve a message with the `shell.fs.path-exist` key:

```
var msg = messageSource.getMessage( "shell.fs.path-exist", … );
```

MessageSourceAccessor

Spring declares another class, `MessageSourceAccessor`,[63] which further facilitates working with `MessageSource` objects. The UML diagram in Figure 3.20 lists the methods.

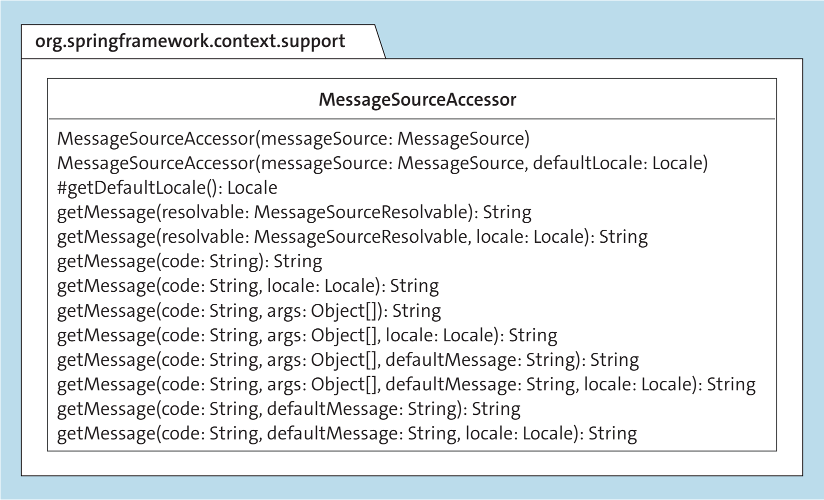

org.springframework.context.support
MessageSourceAccessor
MessageSourceAccessor(messageSource: MessageSource)
MessageSourceAccessor(messageSource: MessageSource, defaultLocale: Locale)
#getDefaultLocale(): Locale
getMessage(resolvable: MessageSourceResolvable): String
getMessage(resolvable: MessageSourceResolvable, locale: Locale): String
getMessage(code: String): String
getMessage(code: String, locale: Locale): String
getMessage(code: String, args: Object[]): String
getMessage(code: String, args: Object[], locale: Locale): String
getMessage(code: String, args: Object[], defaultMessage: String): String
getMessage(code: String, args: Object[], defaultMessage: String, locale: Locale): String
getMessage(code: String, defaultMessage: String): String
getMessage(code: String, defaultMessage: String, locale: Locale): String

Figure 3.20 “MessageSourceAccessor” Class

There are `getMessage(…)` methods that don’t specify the language because if it’s missing, it’s retrieved from `LocaleContextHolder.getLocale()`. This reduces the code volume when using `MessageSource`. If there is no text for the key, the method returns an empty string.

To use `MessageSourceAccessor`, we first need a `MessageSource` object, which we pass in the constructor. It could look like this:

---

[63] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/context/support/MessageSourceAccessor.html*

```
final MessageSourceAccessor messageSource;

public Translater( MessageSource messageSource ) {
  this.messageSource = new MessageSourceAccessor( messageSource );
}
```

Internally, the default language is requested, and it no longer needs to be explicitly specified:

```
var msg = messageSource.getMessage( "shell.fs.path-exist",
                                    new Object[]{ "c:/test" });
```

ReloadableResourceBundleMessageSource

We've worked with the `ResourceBundleMessageSource` class so far, but there is a second class called `ReloadableResourceBundleMessageSource`,[64] which has some advantages. First, the class can load not only properties files but also XML files, and, second, the resources can be reloaded automatically when a cache time expires. The usage can look like this:

```
@Bean
MessageSource messageSource() {
  var messageSource = new ReloadableResourceBundleMessageSource();
  messageSource.setBasenames( "classpath*:messages/",
                              "classpath*:valid/validation" );
  messageSource.setCacheSeconds( 2 );
  messageSource.setDefaultEncoding( "UTF-8" );
  messageSource.setUseCodeAsDefaultMessage( true );
  return messageSource;
}
```

The factory method creates a `MessageSource` implementation, so the auto-configuration won't build a copy. We've already seen three of the methods in `ResourceBundleMessageSource`, but the `setBaseName(…)` method works differently than in `ResourceBundleMessageSource` because files and directories must be specified here; internally, the class is *not* based on `ResourceBundle`.

The `setCacheSeconds(…)` method is new and is used to specify the cache time in seconds. The value -1 means that the resources are cached forever, 0 makes the cache invalid on each call to `getMessage(…)`, which is expensive. All values greater than 0 specify the cache time in seconds.

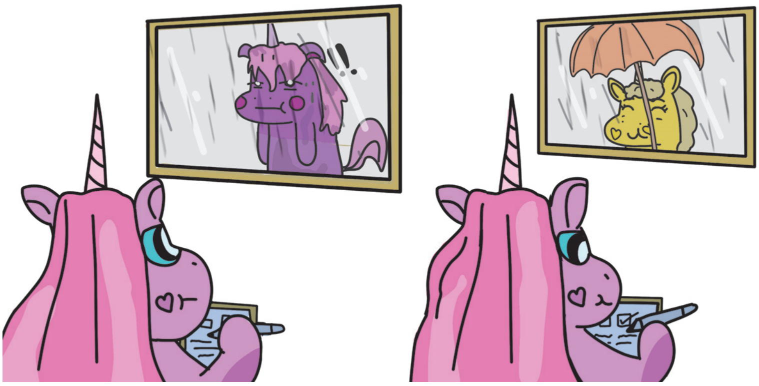
---

[64] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/context/support/ReloadableResourceBundleMessageSource.html*

3.8 Test-Driven Development with Spring Boot

This section of the chapter is dedicated to exploring the subject of application testing, with a specific emphasis on those that operate within the Spring container. We'll be examining the widely used testing frameworks JUnit 5, AssertJ, and Mockito, and analyzing the process of testing injection.

We'll also address testing at various stages throughout the book, particularly when discussing particular technologies such as database access or RESTful web services.



3.8.1 Test Related Entries from Spring Initializr

With the Spring Initializr, we automatically get a dependency with various test libraries.

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-test</artifactId>
  <scope>test</scope>
</dependency>
```

Listing 3.14 pom.xml Snippet

Spring Boot integrates a number of solutions and libraries via `spring-boot-starter-test` to improve the "testing experience":

- *Spring test*
- *JUnit 5* (*http://junit.org/junit5/*)
- *AssertJ* (*https://joel-costigliola.github.io/assertj/*)
- *Hamcrest* (*http://hamcrest.org/JavaHamcrest/*)
- *Mockito* (*http://mockito.org*)
- *JSONassert* (*https://github.com/skyscreamer/JSONassert*) and *JSONPath* (*https://github.com/json-path/JsonPath*)

Note

Spring Boot doesn't include the JUnit Vintage module to support JUnit 4; those with old JUnit 4 test cases will need to manually include the JUnit Vintage module.

Initializr has generated the `Date4uApplicationTests` class in the *src/test/java* directory.

```
import org.junit.jupiter.api.Test;
import org.springframework.boot.test.context.SpringBootTest;

@SpringBootTest
class Date4uApplicationTests  {

  @Test
  void contextLoads() {
  }

}
```

Listing 3.15 Date4uApplicationTests.java

Two annotations can be seen in this class: `@Test` from JUnit and `@SpringBootTest`. What the latter means and why it's useful will be explained later under Section 3.8.9.

3.8.2 Annotation @Test

JUnit will automatically call all methods annotated with `@Test`. This has nothing to do with `@SpringBootTest` because `@SpringBootTest` starts the Spring container. We'll work without `@SpringBootTest` in the first step and add the container later.

Assertions about a state can be taken within these test methods. There are two ways to do this:

1. Using methods of the JUnit-specific class `org.junit.jupiter.api.Assertions`.
2. Methods of the alternative library *AssertJ*, and using the `assertThat(…)` methods with a Fluent API to check certain states.

To get started, let's write our first test using JUnit and AssertJ. If you're already familiar with these frameworks, feel free to skip this section.

3.8.3 Test Case for the FileSystem Class

The `FileSystem` class is compact and is therefore shown here as a reminder.

```
public class FileSystem {
  private final Path root =
    Paths.get( System.getProperty("user.home") ).resolve( "fs" );
  public FileSystem() {
    try { if ( ! Files.isDirectory(root) ) Files.createDirectory(root); }
    catch ( IOException e ) { throw new UncheckedIOException( e ); }
  }

  public long getFreeDiskSpace() {
    return root.toFile().getFreeSpace();
  }

  public byte[] load( String filename ) {
    try { return Files.readAllBytes( root.resolve( filename ) ); }
    catch ( IOException e ) { throw new UncheckedIOException( e ); }
  }

  public void store( String filename, byte[] bytes ) {
    try { Files.write( root.resolve( filename ), bytes ); }
    catch ( IOException e ) { throw new UncheckedIOException( e ); }
  }
}
```

Listing 3.16 FileSystem.java

Create the FileSystemTest Test Class

For a method of the `FileSystem` class, we write a test case that will run independently of the Spring Framework or Spring Boot. The `FileSystem` class is in the `com.tutego.date4u.core` package, so the first thing to do is to create a `FileSystemTest` class under *src/test/java* in the same package—an IDE can do this for us. Usually, we adopt the same package structure because private variables or methods can become package-visible, and a test case then has access without problems. A suffix `Test` is common for test classes. The class also doesn't have to be `public` since JUnit 5.

Write Test Method

The first test is to show that `getFreeDiskSpace()` returns a positive value.

```
@Test
@DisplayName( "free disk space has to be positive" )
void free_disk_space_has_to_be_positive() {

  // given
  FileSystem fileSystem = new FileSystem();

  // when
  long actual = fileSystem.getFreeDiskSpace();
  // then
  // assertTrue( actual > 0, "Free disk space was not > 0" );
  assertThat( actual ).isGreaterThan( 0 );
}
```

Listing 3.17 FileSystemTest.java

The method name is chosen to be meaningful so that it's like a statement, so to speak: `free_disk_space_has_to_be_positive` means that if the test case fails, the disk space wasn't positive after all, but 0 or negative, or perhaps there was an exception. Test method names with underscores are fine. If you don't like that, you can use `@DisplayName` and set a name.

Each test has three stages at its core: the line comments make this clear with `given`, `when`, and `then`. In the first step, a `FileSystem` object is built. The second step calls the method to be tested: `getFreeDiskSpace()`. The third step tests if the effects are as expected: we hope for a positive value. The APIs of JUnit 5 and AssertJ are different, and the following code shows both variants:

- JUnit:

  ```
  assertTrue( actual > 0, "Free disk space was not > 0" );
  ```

- AssertJ:

  ```
  assertThat( actual ).isGreaterThan( 0 );
  ```

The error message is optional with JUnit, but without a message, you're quickly in the dark and wonder what the problem is. With AssertJ, a message is also possible, but the error display is already much more verbose, so we can do without explicit messages. The two lines also show the different approach of the APIs: while in JUnit, what is expected, such as `actual > 0`, is in front, whereas, in AssertJ, what is tested (`actual`) is in front, and the Fluent API formulates our wishes for the values.

Test for Loading and Saving

Next, we'll compose a test that covers the loading and saving functionality. We combine the two `FileSystem` methods `store(…)` and `load(…)`.

```
@Test
void store_and_load_successful() {
  FileSystem fileSystem = new FileSystem();
  fileSystem.store( "test.txt", "Hello World".getBytes() );
  byte[] actual = fileSystem.load( "test.txt" );
  assertThat( actual ).containsExactly( "Hello World".getBytes() );
}
```

Listing 3.18 FileSystemTest.java

Test Exceptions

If something can't be loaded, we expect an exception. The test case could look like this listing.

```
@Test
void load_unknown_file_throws_exception() {
  FileSystem fileSystem = new FileSystem();
  assertThatThrownBy( () -> {
    fileSystem.load( UUID.randomUUID().toString() );
  } ).isInstanceOf( UncheckedIOException.class );
}
```

Listing 3.19 FileSystemTest.java

To test exceptions with AssertJ, not `assertThat(…)` but `assertThatThrownBy(…)` is used. The lambda expression contains the code that throws the exception. The return of `assertThatThrownBy(…)` is of type `AbstractThrowableAssert`, which declares, among other things, `isInstanceOf(…)` for testing whether the exception was really of type `UncheckedIOException`. There are other methods that check, for example, the message in the exception.

Bugfix: Directory Traversal

The `FileSystem` class has a severe security problem in its current implementation. If, on a Windows system, you write

```
FileSystem fileSystem = new FileSystem();
var file = fileSystem.load(
    "../../../../../../../../../../../Windows/notepad.exe" );
```

then in `file`, we would have the `byte` array from the EXE file. The problem lies in the `root.resolve(filename)` expression in the `FileSystem` class: we could completely escape from `root` with `..` and, therefore, hop everywhere. This security problem is called *directory traversal*.

Fortunately, the issue can be easily resolved by modifying the `load(…)` and `store(…)` methods. Instead of using the `resolve(…)` method directly, we can create a separate method for it.

```
public byte[] load( String filename ) {
  try {
    Path path = resolve( filename );
    return Files.readAllBytes( path );
  }
  catch ( IOException e ) { … }
}
```

Listing 3.20 FileSystem.java

The custom `resolve(…)` method proceeds by continuing to resolve the path on the `root` object with the `Path` method `resolve(…)`, but then checking in the next step if the prefix is still `root`. If it's not, then someone has diverted away from `root`, and this should result in a `SecurityException`.

```
private Path resolve( String filename ) {
  Path path = root.resolve( filename ).toAbsolutePath().normalize();
  if ( ! path.startsWith( root ) )
    throw new SecurityException( "Access to " + path + " denied" );
  return path;
}
```

Listing 3.21 FileSystem.java

A test method can show that.

```
@Test
void load_arbitrary_file() throws IOException {
  FileSystem fileSystem = new FileSystem();
  assertThatThrownBy( () -> {
    fileSystem.load( "../../../../../../../../../../../Windows/notepad.exe" );
  } ).isInstanceOf( SecurityException.class )
      .hasMessageContaining( "Access to" )
      .hasMessageContaining( "denied" );
}
```

Listing 3.22 FileSystemTest.java

Of course, the test cases can be extended further, but this example provides you with a good introduction to automated test cases.

The next thing we want to look at is wired components that are to be replaced for testing. We may still find a problem in the `FileSystem` class.

3.8.4 Test Multitier Applications, Exchange Objects

Static methods are easy to test, provided they have no side effects and simply return something: some value comes in, a value comes out, and the test checks the return. Most of the time, however, a test isn't that simple:

- Often one object needs many other objects, so there are many dependencies.
- Background operations can be triggered that can be potentially expensive. For example, with `PhotoService`, the `FileSystem` is involved, and whenever we use the file system, a test has to clean up after the run.

This highlights the fundamental contrast between integration tests and unit tests. See Figure 3.21 for a refresher on the dependencies of `PhotoService`.

`PhotoService` references two other services: a `Thumbnail` implementation on one side, for example, implemented by `AwtBicubicThumbnail`, and on the other side the `FileSystem`, which operates on the local file system.

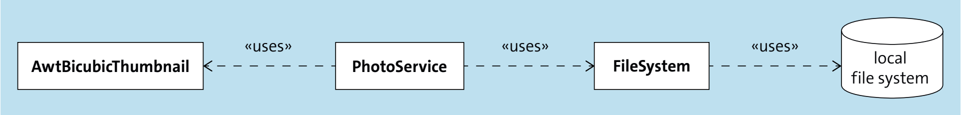


Figure 3.21 Dependencies on "PhotoService"

The goal is to build a test case for `PhotoService` in such a way that a local file system isn't necessary. A test should have reduced side effects and not perform real read/write operations on the file system. This is also true for other systems: we don't want to write to the database, send SMS and emails, and so on. The goal is an isolated component test, but not an integration test with multiple layers.

Testing components is a crucial step in ensuring that the system works smoothly. Fortunately, the process of testing components that access the infrastructure isn't as complicated as it may seem. It involves swapping out the actual components of the infrastructure and replacing them with imitation objects, known by different names such as *dummy*, *stub*, *fake*, or *mock* objects. These mimics are designed to replicate the behavior of the actual infrastructure components, allowing developers to test the system without affecting the infrastructure.

One of the most common types of mimic is a dummy object. It's similar to a crash test dummy used in the automotive industry. A dummy object has the same weight and dimensions as a real object, but isn't functional. In the context of software development, a dummy object is designed to simulate the behavior of a particular component. For example, if a component of the infrastructure is responsible for handling user input, a dummy object would be created to simulate user input without actually affecting the system. It's important to note that while a dummy object may look like the real thing from a distance, it's not an exact replica. It's a tool used for testing purposes and doesn't have the same functionality as the actual component. Nonetheless, by using dummy objects and other types of mimics, developers can test the system in a controlled environment and identify any potential issues before the system goes live.

A `FileSystem` dummy can mimic a real file system implementation, and we can feed our `PhotoService` with this dummy, and the original `FileSystem` won't be referenced at all.

Suppose there is a `DummyFileSystem` class that has the same API as a `FileSystem`; then we could push the dummy to the `PhotoService`, and the `PhotoService` wouldn't notice (see Figure 3.22).

Whether or not to create a dummy for the thumbnail is subjective. However, because the implementation doesn't produce any side effects, we'll adhere to the original approach.

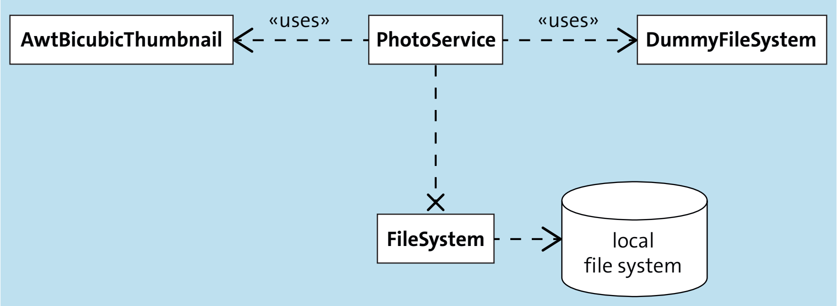

AwtBicubicThumbnail
«uses»
PhotoService
«uses»
DummyFileSystem
FileSystem
local
file system

Figure 3.22 Imitated and Faked: “PhotoService” Using a Dummy

As a reminder, `PhotoService` needs a `FileSystem` and a `Thumbnail` implementation via the constructor injection.

```
@Service
public class PhotoService {

  private final FileSystem fs;
  private final Thumbnail  thumbnail;

  public PhotoService( FileSystem fs, Thumbnail thumbnail ) {
    this.fs = fs;
    this.thumbnail = thumbnail;
  }

  public String upload( byte[] imageBytes ) {
    …
    fs.store( … )
    thumbnail.thumbnail( … )
    …
  }

  …
}
```

Listing 3.23 PhotoService.java

Write a Test Case for PhotoService

In the test case, we can recreate this and call the constructor ourselves. A complete test could look like the following listing.

```
// @SpringBootTest
class PhotoServiceTest {
  private static final byte[] MINIMAL_JPG = Base64.getDecoder().decode(
     "/9j/4AAQSkZJRgABAQEASABIAAD/2wBDAP////////////////////////////////////"
    +"//////////////////////////////////////////////////wgALCAABAAEBAREA/8QA"
    +"FBABAAAAAAAAAAAAAAAAAAAAAP/aAAgBAQABPxA=" );    // https://git.io/J9GXr

  private static class DummyFileSystem extends FileSystem {
    @Override public long getFreeDiskSpace() { return 1; }
    @Override public byte[] load( String filename ) { return MINIMAL_JPG; }
    @Override public void store( String filename, byte[] bytes ) { }
  }

  @Test
  void successful_photo_upload() {
    // given
    FileSystem fs = new DummyFileSystem();
    PhotoService photoService=new PhotoService(fs,new AwtBicubicThumbnail());

    // when
    String imageName = photoService.upload( MINIMAL_JPG );

    // then
    assertThat( imageName ).isNotEmpty();
  }
}
```

Listing 3.24 PhotoServiceTest.java

To begin, it's worth noting that `@SpringBootTest` isn't required because Spring's testing infrastructure won't be used. Class `DummyFileSystem` is created via a subclass of `FileSystem`; in the code, it's done as a static, nested class. As a subclass, it has all the methods of `FileSystem`, that is, `getFreeDiskSpace()`, `load(…)`, and `store(…)`. `DummyFileSystem` overrides these methods and returns predefined values:

- `getFreeDiskSpace()` always returns 1.
- Loading always returns a small JPEG.[65] An empty `byte[]` in the return would causes problems later when the result of `load(…)` is the source for the preview image. The `byte[]` array shouldn't be modified by the client.
- Nothing happens on `store(…)`, and the method returns `void` anyway, so the code is shortest.

The actual test case is simple in structure. We stick to the three steps: *given*, *when*, *then:*

1. *Given*: We replay Spring's constructor injection and build the `PhotoService` by passing instances of `DummyFileSystem` and `AwtBicubicThumbnail` in the constructor.

---

[65] The small JPEG is Base64 encoded; the original can be found at *https://git.io/J9GXr*.

2. *When*: On `PhotoService`, the `upload(…)` method is called with the bytes from the minimum JPG.
3. *Then*: Finally, the test checks the states: the return image name must not be empty. For demonstration, this simple query is sufficient.

The manual implementation of dummy classes is time-consuming and can be automated. Just think of interfaces with very many methods—then a dummy implementation quickly leads to a lot of code.

3.8.5 Mocking Framework Mockito

Creating dummy classes manually can be a tedious and time-consuming process, but it can be simplified thanks to *mocking libraries*. These libraries provide tools to create *mock* objects, which are proxies that mimic the behavior of a particular interface or component. Mock objects are created during the testing phase and are given the desired behavior to simulate the real object's actions.

Mockito (*https://site.mockito.org*) is one of the most widely used mocking libraries. The library allows developers to define the behavior of the mock object and specify the conditions under which the mock object should behave in a certain way. Mockito also provides the ability to verify that certain actions have been taken on the mock object. This can help identify any issues in the system before it's released to the public, ensuring a high level of quality and reliability. The Spring Boot Starter Test dependency includes Mockito, making it easy to integrate the library into a Spring Boot project.

The next sections show small examples of Mockito's API, isolated without Spring Boot. Finally, we transfer the knowledge into a Spring component test.

Mockito generates proxy objects, and there are different approaches to building them. The first one uses static method `org.mockito.Mockito.mock(…)`:

```
FileSystem fileSystem = Mockito.mock( FileSystem.class );
```

A type token is passed, and the result is "something of type `FileSystem`." Mockito uses different strategies for generation; in our case, bytecode is built at runtime for the `FileSystem` class. How does the mock react when methods are called?

```
Logger log = LoggerFactory.getLogger( getClass() );

log.info( fileSystem.toString() );
// […] Mock for FileSystem, hashCode: 1373949107

log.info( "free disk space: {}, load: {}",
          fileSystem.getFreeDiskSpace(), fileSystem.load( "" ) );
// […] free disk space: 0, load: null

fileSystem.store( "", MINIMAL_JPG );
```

Mock objects are essentially empty shells that simulate the behavior of a particular component or interface during testing. When a method is called on a mock object, nothing happens, and no actual work is performed. This allows developers to test the behavior of their code without the need for a fully functional component.

Mock objects are sometimes referred to as *null objects* because they exist, but they don't provide any functionality or behavior. They are used solely for testing purposes and can be easily integrated into a testing environment.

In addition to simulating the behavior of a component or interface, mock objects also return default values when a method is called.

Stubbing the Mock Object

The `FileSystem` object can be used in the test case, and it's similar to the `DummyFileSystem` object. Only the old dummy object had an advantage: it showed a special behavior, for example, by returning a certain byte array when loading or answering with `1` when asked for the number of free bytes.

A mock object can also be configured to respond with certain return, which is called *stubbing*. In the code, it looks like this:

```
FileSystem fileSystem = mock( FileSystem.class );
given( fileSystem.getFreeDiskSpace()  ).willReturn( 1L );
given( fileSystem.load( anyString() ) ).willReturn( MINIMAL_JPG );
```

Expression `given(MOCK METHOD).willReturn(RETURN)` determines in the first case that the `getFreeDiskSpace()` method should respond with `1` and that `load(…)` can be called with any string and should return the mini-JPG. The methods `given(…)` and `anyString()` are statically imported from `org.mockito.BDDMockito.*` for the example.

The log outputs show the following:

```
log.info( "free disk space: {}, load: {}",
        fileSystem.getFreeDiskSpace(), fileSystem.load( "" ) );
// […] free disk space: 1, load: […]
fileSystem.store( "", MINIMAL_JPG );
```

In summary, the initial step involves creating an empty shell of a component. Subsequently, the next step entails defining the desired behavior for the component. The new mock `FileSystem` object is capable of entirely replacing the previous implementation (`DummyFileSystem`), which is an encouraging development.

If we transfer the snippets into a test class, it might look like this:

```
class PhotoServiceTest {
  private static final byte[] MINIMAL_JPG = …
  private final Logger log = LoggerFactory.getLogger( getClass() );
  @Test
  void successful_photo_upload() {
    FileSystem fileSystem = mock( FileSystem.class );
    given( fileSystem.getFreeDiskSpace()  ).willReturn( 1L );
    given( fileSystem.load( anyString() ) ).willReturn( MINIMAL_JPG );
   // Implementation from test case as known
}
```

The variable `fileSystem` is local in the test. However, an instance variable can also be used.

@Mock, MockitoExtension

The mock object has so far built the static `mock(…)` method. This is a form of "fetching," but there is another way: Mockito can initialize the instance variables annotated with `@Mock`[66] with a mock object. For this to be possible, the test case must be annotated with `@ExtendWith` and configured with the Mockito extension.[67]

```
@ExtendWith( MockitoExtension.class )
// @MockitoSettings( strictness = Strictness.LENIENT )
class PhotoServiceTest {
  private static final byte[] MINIMAL_JPG = …
  private final Logger log = LoggerFactory.getLogger( getClass() );

  @Mock FileSystem fileSystem; // = mock( FileSystem.class );
  // @Mock(strictness = Strictness.LENIENT) FileSystem fileSystem;

  @BeforeEach
  void setupFileSystem() {
    given( fileSystem.getFreeDiskSpace()  ).willReturn( 1L );
    given( fileSystem.load( anyString() ) ).willReturn( MINIMAL_JPG );
  }

  // Mocked FileSystem is ready in the test methods
}
```

`MockitoExtension` recognizes instance variables annotated with `@Mock` and initializes them. Conceptually—as can be seen in the comment—this is like calling the static `mock(…)` method.

---

[66] *https://javadoc.io/doc/org.mockito/mockito-core/latest/org/mockito/Mock.html*

[67] *https://javadoc.io/doc/org.mockito/mockito-junit-jupiter/latest/org/mockito/junit/jupiter/MockitoExtension.html*

Note

If a mock object is configured with a behavior, the stubbed methods must also be called. If this isn't desired, `@Mock` must have the annotation attribute `strictness` set to `Strictness.LENIENT`. This means something like, "Stay *lenient*; it's okay if certain things are configured but not used in the test." This can also be assigned for the classes via type annotation `@MockitoSettings`.

The JUnit test case performs the stubbing in the `setupFileSystem()` method annotated with `BeforeEach`; the annotation type comes from Junit 5 and means that `setupFileSystem()` should be called before each test case. We can, of course, build and stub the mock locally, but that's a matter of taste.

@Spy

In addition to the `@Mock` annotation type, let's learn about another Mockito annotation, `@Spy`,[68] which configures a "spy" that wraps around the real object. Later, we can ask the spy, "What did you hear: what methods were called?" So, while a mock object is entirely new and parameterized by us, an `@Spy` wraps the original: The proxy lets all method calls through and, in the background, remembers which methods were called how often with which arguments.

An `@Spy` is useful for our `AwtBicubicThumbnail` component. A mock isn't useful, but an `@Spy` is because if `PhotoService` accesses the thumbnail object later, we can ask the spy if certain `Thumbnail` methods were really called. In code, an instance variable is annotated first:

```
@Spy AwtBicubicThumbnail thumbnail;    // or Thumbnail
```

Then, `PhotoService` can access a mock object of type `FileSystem` and a spy of type `AwtBicubicThumbnail`:

```
PhotoService photoService = new PhotoService( fileSystem, thumbnail );
```

In the last line, it's easy to see that we're building the instance ourselves, so we're replaying the IoC manually.

3.8.6 @InjectMocks

Mockito can also build objects and perform a simple form of wiring. Annotation type `@InjectMocks` is used for this purpose.[69] The instance variables would look like this:

---

[68] *https://javadoc.io/doc/org.mockito/mockito-core/latest/org/mockito/Spy.html*
[69] *https://site.mockito.org/javadoc/current/org/mockito/InjectMocks.html*

```
@Mock FileSystem          fileSystem;
@Spy  AwtBicubicThumbnail thumbnail;
@InjectMocks PhotoService photoService;
```

The use of `@InjectMocks` is a bit like `@Autowired`: Mockito creates a `PhotoService` object and calls the parameterized constructor with the built mock and spy objects. However, Mockito isn't a complete dependency injection container: objects not managed by Mockito aren't created, and then the variables remain `null`.

Mockito not only tries to call the constructor but also calls possible setters or even set instance variables if they match the mock or spy types. What is called in which order is explained in more detail in the API documentation at *https://site.mockito.org/javadoc/current/org/mockito/InjectMocks.html*.

3.8.7 Verify Behavior

Test method `successful_photo_upload()` doesn't pay attention to what happens externally, it only checks the return, that is, the string, to make sure it's not empty. `successful_photo_upload()` doesn't check whether `PhotoService` really talks to the file system. The test case could well do that in two steps:

1. Upload a photo.
2. Ask the file system if a new file and a thumbnail image has been created.

However, because we replaced the file system with a mock, there are no files.

There is another way, however. We know that the `upload(…)` method talks to `FileSystem` and to the `Thumbnail` implementation; both are Mockito objects. Mockito provides a special `verify(…)` method; the API is similar to the `given(…)` method, only it's used for verification. Here's an example: the test method should check if the method `store(…)` was called twice on `FileSystem` and if method `thumbnail(…)` was called once on the `Thumbnail` implementation:

```
@Test
void successful_photo_upload() {
  String imageName = photoService.upload( MINIMAL_JPG );
  assertThat( imageName ).isNotEmpty();
  verify( fs, times( 2 ) ).store( anyString(),
                                  any( byte[].class ) );
  verify( thumbnail ).thumbnail( any( byte[].class ) );
}
```

The `verify(…)` method is passed:

- The kinds of objects interrogated include an `fs` (type `FileSystem`) and a `thumbnail` (type `Thumbnail`).

- The object returned by `verify(…)` is followed by the condition that must hold: Was the `store(…)` method called with any file name (`String`) and any `byte[]`? Was the `thumbnail(…)` implementation called with any `byte[]`?

You could look at the file names more closely with an argument matcher, but the example makes it clear that Mockito records the method calls on the mock and spy, and `verify(…)` relies on that recording to check our conditions.

Note

Mockito counts all calls. Because the `upload(…)` method calls the `store(…)` method twice (once for the original image and once for the thumbnail image), the count must be specified via `verify( …, times(2) )`.

3.8.8 Testing with ReflectionTestUtils

`FileSystem` has a parameterized constructor. In principle, nonprivate constructors and setters can be easily called in code manually, and this is an advantage when testing.

With field injection, this doesn't look so good because usually the instance variables are private. For a test case to read and write private variables, a program must rely on reflection. (The same is necessary for private constructors or setters). Handcrafting isn't necessary because Spring declares a class `ReflectionTestUtils`[70] with useful static methods:

- `Object getField(Class<?> targetClass, String name)`
- `Object getField(Object targetObject, Class<?> targetClass, String name)`
- `Object getField(Object targetObject, String name)`
- `Object invokeGetterMethod(Object target, String name)`
- `<T> T invokeMethod(Class<?> targetClass, String name, Object... args)`
- `<T> T invokeMethod(Object targetObject, Class<?> targetClass, String name, Object... args)`
- `<T> T invokeMethod(Object target, String name, Object... args)`
- `void invokeSetterMethod(Object target, String name, Object value)`
- `void invokeSetterMethod(Object target, String name, Object value, Class<?> type)`
- `void setField(Class<?> targetClass, String name, Object value)`
- `void setField(Class<?> targetClass, String name, Object value, Class<?> type)`
- `void setField(Object targetObject, Class<?> targetClass, String name, Object value, Class<?> type)`
- `void setField(Object targetObject, String name, Object value)`
- `void setField(Object targetObject, String name, Object value, Class<?> type)`

---

[70] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/test/util/ReflectionTestUtils.html*

Change the FileSystem Path for Testing

We'll now redirect our attention to the `FileSystem` class and, specifically, the initialization of the `root` variable:

```
@Component
public class FileSystem {

  private final Path root =
      Paths.get( System.getProperty( "user.home" ) ).resolve( "fs" );

  …
}
```

The instance variable `root` is automatically assigned by the constructor.

Suppose you want to select a different directory in a test case. One option is to use `ReflectionTestUtils` to reassign the `root` variable so that it's set to a different `Path` object. `Path` objects not only refer to other directories, but there is always a whole file system behind them. It would be interesting to use an in-memory file system such as *Jimfs* (*https://github.com/google/jimfs*), so that the test case of `FileSystem` can take place completely in memory.

To set `root` to a different path via `ReflectionTestUtils`, the final keyword would first have to be dropped; private is allowed for the variable. In the test case, `ReflectionTestUtils` can assign a new `Path` to the `root` variable:

```
Path path = …
FileSystem filesystem = new FileSystem();
ReflectionTestUtils.setField( filesystem, "root", path );
```

The problem with the variant is that it doesn't really help, and we have a serious testing problem. The problem is that the parameterless constructor still initializes the `root` variable with the local data system and also creates an *fs* subdirectory in case it doesn't exist. `ReflectionTestUtils` runs after the constructor call. That is, an *fs* directory was created in the user directory, not in the directory that was later assigned by reflection.

In other words, just because `ReflectionTestUtils` can write private variables for testing doesn't mean that the test can be validly implemented. The current implementation of `FileSystem` highlights a design problem: the parameterless constructor must not initialize a variable and build the directory. This is the purpose of test-driven design: testability shapes the design and the API.

A better solution is for a parameterized constructor to accept a `Path` object from outside and store the `Path`:

```
public FileSystem() {
  this( Paths.get( System.getProperty( "user.home" ) )
             .resolve( "fs" ).toAbsolutePath().normalize() );
}

FileSystem( Path root ) {
  this.root = root;
  …
}
```

The advantage is that the test can parameterize the path from the outside. This parameterized constructor doesn't necessarily have to be public. It's perfectly fine if it remains a package-visible constructor because it's only used in the test case.

Tip

Sometimes we don't realize if the component has been developed testably until we write the test cases. If not, we should rewrite the class to fix that.

3.8.9 With or Without Spring Support

In the previous section, we looked at test cases that ran independently from the Spring Framework. However, there are certain benefits that the Spring Framework provides that we may not want to reprogram from scratch. For instance, Spring can load properties, automatically configure objects, initialize proxy objects (e.g., for security testing), and more. There comes a point where recreating the entire setup becomes impractical, and this is where the Spring context comes in handy. This marks the transition from a component test to an integration test, where we can use the Spring context to test our components in a more realistic environment.

Annotation @SpringBootTest

If the `@SpringBootTest` annotation is attached to the test class, this causes the container to start. By default, all components from the main configuration are recognized, and instances can be injected into the test class with `@Autowiring`.

In addition, event handling runs through the runners, which starts the shell—we don't want an interactive shell in the test case. Configurations are also evaluated, and configuration files such as *application.[properties|yml]* are read. Of course, configurations can also be injected into the test class via `@Value`.

The annotation also tells Spring Boot to look for a class annotated with `@SpringBootConfiguration` for the main configuration; this is our `@SpringBootApplication`.

Lifecycle

`@SpringBootTest` starts and fills the context only once in the test case and performs the injections only once. The objects injected into this test class with `@Autowired` persist through each run.

After initialization, the test methods are called. The lifecycle is shown in Figure 3.23.

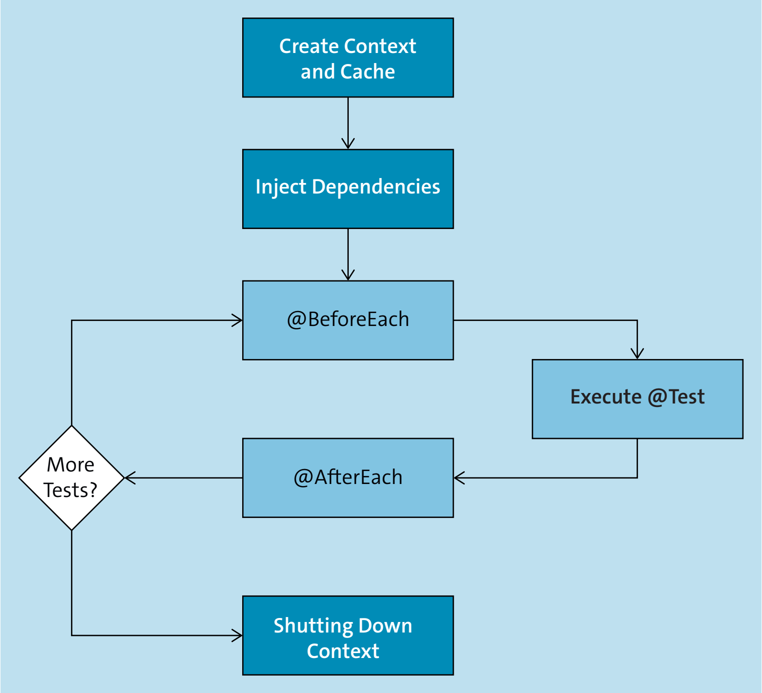


Figure 3.23 Lifecycle of a Test Case

JUnit 5 first calls the `@BeforeEach` annotated method, then the test method, and then an `@AfterEach` callback method; the methods don't have to be present. The figure uses two levels of brightness, expressing what is part of Spring and what is part of normal JUnit test cases. The figure also makes it clear that the container is only started once, and the individual test cases within that class work with that one initialized container. At the very end, when all test cases have been processed, the container is closed and shut down in the regular way.

This has advantages and disadvantages. The advantage is a better runtime performance of all tests: if all tests are executed with only one context, the context doesn't have to be rebuilt again and again for each method. The disadvantage is that the tests don't run completely isolated in case test methods change something in the context. We'll discuss this later in Section 3.8.14 where you'll see that Spring can also be configured differently.

Improve Test Time, Minimize Loaded Classes

Component tests are typically focused on testing a single class. When the container initializes all components, it can become too extensive and obscure the real dependencies of the component being tested. Additionally, longer startup times and extensive logging of all components can make troubleshooting more difficult. It's important for test cases to not only be processed automatically but also quickly. While a performance loss may not be significant with only 10 test cases, it can become an issue with hundreds or thousands of tests. Refactoring can be a challenging task if this isn't addressed early on.

A test case should be limited to only the types that are needed in the test case. For this purpose, an array of only those classes that are desired in a test case are assigned to annotation attribute `classes` in `@SpringBootTest`. In the following, for example, only classes `A` and `B` are considered:

```
@SpringBootTest( classes = { A.class, B.class } )
```

In addition to improving test time, another benefit is that types are better kept in view that are needed for the test case, and things that are used in the background by the class under test aren't overlooked.

@SpringBootTest Annotation Type

`SpringBootTest` has a number of meta-annotations:

```
@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@BootstrapWith(SpringBootTestContextBootstrapper.class)
@ExtendWith({SpringExtension.class})
public @interface SpringBootTest { … }
```

`SpringExtension`[71] integrates the Spring testing framework into the JUnit 5 test environment. In older code, you can sometimes find the line `@ExtendWith(SpringExtension.class)`, but it's no longer necessary.

`@SpringBootTest` declares the following annotation attributes:

```
public @interface SpringBootTest {

  @AliasFor("properties")
  String[] value() default {};

  @AliasFor("value")
```

---

[71] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/test/context/junit/jupiter/SpringExtension.html*

```
  String[] properties() default {};

  String[] args() default {};

  Class<?>[] classes() default {};

  WebEnvironment webEnvironment() default WebEnvironment.MOCK;

  enum WebEnvironment { … }
}
```

We've just seen the `classes` annotation attribute; let's now deal with `properties`, which is an alias for `value`.

3.8.10 Test Properties

Spring Boot also automatically loads the two configuration files *application.properties* and *application.yml* in the test case. All configuration properties that are declared are also available in the test case.

Often in a test case, properties are assigned differently or additional properties are desired. There are two ways to add or reassign properties.

One way is to specify an array of key-value pairs in `@SpringBootTest` with the help of the `properties` annotation attribute:

```
@SpringBootTest( properties = { "firstProperty=firstValue",
                                "secondProperty=secondValue" } )
```

The annotation attribute `properties` is an alias for `value`.

A second possibility uses another annotation `TestPropertySource`. It also allows assignment to annotation attribute `properties` and lets you specify properties files that will be loaded. We'll look at this directly after the task.

Disable Shell

When a test case starts and the Spring Shell is included, you notice that the shell automatically “hangs.” This is because the shell is a kind of runner, which is also started in the test case. Configuration property `spring.shell.interactive.enabled` can be set to `false`, which disables the shell. This is what we want to do for the test case. The notations are equivalent, which disable the shell.

```
// @SpringBootTest( properties = {"spring.shell.interactive.enabled=false"} )
// @SpringBootTest( properties = "spring.shell.interactive.enabled=false" )
// @SpringBootTest( value = "spring.shell.interactive.enabled=false" )
@SpringBootTest( "spring.shell.interactive.enabled=false" )
```

```
class Date4uApplicationTests {
  …
}
```

Listing 3.25 Date4uApplicationTests Extension

3.8.11 @TestPropertySource

Annotation `@TestPropertySource`[72] can be used not only to declare properties but also to specify files; the annotation is repeatable. A file is specified by the `locations` annotation attribute, which is an alias for `value`.

Here's an example:

```
@SpringBootTest
@TestPropertySource( locations = "classpath:test.properties" )
// @TestPropertySource( "classpath:test.properties" )
@TestPropertySource( properties = "spring.shell.interactive.enabled=false" )
class …
```

So, the commented-out notation is a shorter alternative if only files are to be loaded. The example also shows how the annotation attribute `properties` can be used to turn off the shell. That is, to set properties we have, in principle, two possibilities:

- Variant 1 via `@SpringBootTest`:

  ```
  @SpringBootTest( properties = "spring.shell.interactive.enabled=false" )
  ```

- Variant 2 via `@TestPropertySource`:

  ```
  @TestPropertySource( properties = "spring.shell.interactive.enabled=false" )
  ```

There are Spring tests where `@SpringBootTest` isn't used; these are slice tests, such as for database access code or controllers. Then, it's useful that you can also set `properties` for `TestPropertySource`.

Task: Test Properties

If you want to practice understanding the priority of test properties, you can solve the following task:

- **Part 1**
  - In the *application.properties* file, set property `date4u.filesystem.minimum-free-disk-space`. ✓ (We've already done that.)
  - Inject the property assignment into an instance variable in the test class with `@Value`.
  - Output the value in a test method.

---

[72] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/test/context/TestPropertySource.html*

- **Part 2**
  - Create a *test.properties* file in the (correct) resource directory.
  - Set property `date4u.filesystem.minimum-free-disk-space` to another value.
  - Include `@TestPropertySource` in the test class, and set the file name.
  - Test that the assignment is different.

Suggested solution: First, we put *test.properties* in the *src/test/resources* directory.

```
date4u.filesystem.minimum-free-disk-space=200000
```

Listing 3.26 test.properties

Next, in the `Date4uApplicationTests` class that was created from the Initializr, we set some code in the `contextLoads()` method that was empty until now.

```
@SpringBootTest
@TestPropertySource( locations  = "classpath:test.properties",
                     properties = "spring.shell.interactive.enabled=false" )
class Date4uApplicationTests {

  private final Logger log = LoggerFactory.getLogger( getClass() );

  @Value( "${date4u.filesystem.minimum-free-disk-space}" )
  private long minimumFreeDiskSpace;

  @Test
  void contextLoads() {
    log.info( "{}", minimumFreeDiskSpace );
  }
}
```

Listing 3.27 Date4uApplicationTests

3.8.12 @ActiveProfiles

In regular Spring applications, the `spring.profiles.active` property sets the profiles. This isn't true for test cases, however, where the profiles are activated by annotation `@ActiveProfiles`:[73]

```
@ActiveProfiles( "profileA" )
@ActiveProfiles( { "profileA", "profileB" } )
```

Depending on the set profiles, certain beans can, of course, be created again.

---

[73] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/test/context/ActiveProfiles.html*

3.8.13 Appoint Deputy

Let's recall the relationship between `PhotoService`, `Thumbnail`, and `FileSystem`, as depicted in Figure 3.24.

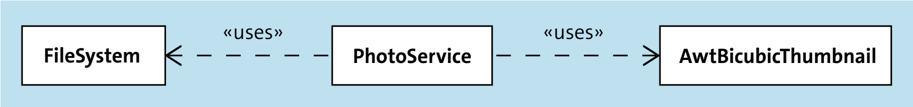


Figure 3.24 "PhotoService" Using a "Thumbnail" Implementation and "FileSystem"

`PhotoService` relies on `FileSystem` to load and store photos, and it also relies on a `Thumbnail` implementation. We've written a parameterized `PhotoService` constructor that assumes a `FileSystem` and a `Thumbnail`. In principle, we could also have taken a field injection or setter injection—the only important thing is that the Spring Framework independently performs wiring during initialization.

If we use the Spring container, we have a small problem because Spring is overzealous and automatically does all the wiring. For the test, however, we need to break the initialization chain and tell Spring to use a dummy `FileSystem` implementation, for example. Three possible solutions will be presented next.

Option 1: @TestConfiguration, @Bean, and @Primary

The first option is to write a configuration class that creates a `FileSystem` implementation via an `@Bean` method. The configuration class can be nicely written as a nested type:

```
@SpringBootTest
public class PhotoServiceTest {

  @TestConfiguration
  static class Config {
    @Bean
    @Primary
    FileSystem getFileSystem() {
      return new DummyFileSystem();
    }
  }

  @Autowired private PhotoService photoService;
  …
}
```

The `@TestConfiguration` class is usually included in the actual test class as a nested static type. The special feature of the factory method is that `@Primary` makes the dummy `FileSystem` the main component.

The factory method uses our old `DummyFileSystem` from Section 3.8.4. Of course, a mock object could have been configured by Mockito.

This approach has weaknesses. The first is that we have to write a factory method, which is overhead; previously, we've set `@Mock`, and that was much more compact. Another problem occurs when the existing `FileSystem` is already a `@Primary`. So, this option isn't the best solution overall.

Option 2: SpringBootTest(classes=...)

The idea of the second solution is based on excluding the original `FileSystem` component so that we can omit the `@Primay`:

```
@SpringBootTest( classes = {
    PhotoService.class,
    AwtBicubicThumbnail.class,
    PhotoServiceTest.Config.class,
} )
public class PhotoServiceTest {
  …
  @TestConfiguration
  static class Config {
    @Bean
    public FileSystem getFileUploader() {
      return new DummyFileSystem();
    }
  }
  …
}
```

In `classes`, the specific classes are enumerated that are desired in the test case. Components, for which the test class builds a dummy, won't come up at all, so the original `FileSystem` class is missing from `classes`, and it remains only `PhotoService` and `AwtBicubicThumbnail`. The test configuration is also a component and must be listed.

The `@Primary` annotation is no longer necessary because there are no more conflicts. Of course, we could have used built-up mock objects in the factory method as well.

Options 1 and 2 are fine, but the code volume is too high; with Mockito's mock objects, it must be even shorter.

Option 3: @MockBean for FileSystem

Spring Boot—but not the Spring Framework—declares the annotation type `@MockBean`,[74] which makes the code very compact.

---

[74] *https://docs.spring.io/spring-boot/docs/current/api/org/springframework/boot/test/mock/mockito/MockBean.html*

```
@SpringBootTest
public class PhotoServiceTest {

  @MockBean  private FileSystem   fileSystem;
  @SpyBean   private Thumbnail    thumbnail;
  @Autowired private PhotoService photoService;

  @BeforeEach void setup() { … }

  @Test
  void successful_photo_upload() { … }
}
```

Listing 3.28 PhotoServiceTest.java Extension

With `@MockBean`, Spring creates a mock object, comparable to `@Mock`, the only difference is the following: (1) `@MockBean` builds a mock object; (2) registers it in as a Spring-managed bean, which means it can then be wired; and (3) displaces all objects that are of the same type as the mock. There is no need to set annotation attribute `classes` at `@SpringBootTest`; the original `FileSystem` doesn't exist in this test case.

Note

`@SpyBean` is the Spring counterpart to the `@Spy` annotation and behaves like `@MockBean` to `@Mock`. Don't make a mistake here: `@MockBean` and `@SpyBean` must be used for Spring Boot tests, whereas `@Mock` and `@Spy` come from Mockito and aren't recognized by a Spring Boot test.

3.8.14 @DirtiesContext

We've already discussed the feature that the container and context only start once in a test case. It may be that the individual test methods modify the context, and this "polluted" context passes from one test method to the other test method, which is bad.

Spring provides the annotation `@DirtiesContext`,[75] which marks a test as leaving the context "dirty." The annotation has the effect that the container always completely starts for each test method. This has its price, and the tests are slower. Therefore, this feature should only be used when necessary.

The annotation can be set on the test class or on the individual test methods. Furthermore, there is an annotation attribute that specifies when the context should be restarted.

---

[75] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/test/annotation/DirtiesContext.html*

3.9 Testing Slices Using a JSON Example *

We've seen that Spring builds the entire component tree at `@SpringBootTest`, which is an expensive operation. Actually, all components are never needed in a test case. The `classes` attribute can be used to enumerate the types and reduce the number of classes.

Spring offers another way, called *test slices*, to test with fewer types. To accomplish this, `@SpringBootTest` is replaced by another annotation, for example:

- `@JsonTest`
- `@JdbcTest`
- `@DataJpaTest`
- `@WebMvcTest`

These "slice tests" only build a subset of the component tree with just those components that are currently needed for the scenario.

`@JsonTest` is used when only JSON mappings are tested. `@JdbcTest` and `@DataJpaTest` are responsible for database-oriented repository tests; `@WebMvcTest` is used when a program wants to test controllers; and so on.

There are many `@*Test` annotation types, and you can see alist of them at *https://docs.spring.io/spring-boot/docs/current/reference/html/test-auto-configuration.html*.

Note

Only one `@*Test` annotation can be used at a time. In practice, it makes little sense to perform a controller test on one side and repositories at the same time in only one test class. For this you'll always write two test classes.

3.9.1 JSON

We'll see an example of a `@JsonTest` later, but at this point, let's take a small step back and look at the JSON document format. JSON documents look like this:

```
{
  "data": [
    { "x": "2020-01", "y": 100 },
    { "x": "2022-02", "y": 300 }
  ]
}
```

Objects in JSON are written with curly braces. Objects have properties, which are key-value pairs. The root object in the example has key `data` and is associated with an array. There are subobjects in the array. The two child objects have two members: `x` and `y`.

JSON documents are common for several reasons:

- People can easily read, write, and edit JSON.
- There are JSON libraries for probably every programming language.
- A JavaScript interpreter can convert JSON directly into an object.
- JSON documents are generally smaller than XML documents.
- Programs can easily process JSON; the format is simple (*www.json.org/json-en.html*).

3.9.2 Jackson

There are a number of JSON libraries, but Spring uses the *Jackson* library by default (*https://github.com/FasterXML/jackson*). There are different ways of working. Conveniently, Jackson can map between Java objects and JSON objects in both directions, which is called *marshaling*.[76]

The mapping can be controlled more precisely via annotations. For example, `@JsonIgnore` can hide certain branches, `@JsonProperty` can rename branches, and `@JsonFormat` can specify date-time formats individually. The Javadoc at *https://javadoc.io/doc/com.fasterxml.jackson.core/jackson-annotations/latest/index.html* lists the annotation types with explanations.

Integrate Jackson

The Jackson library isn't part of the core Spring Boot Starter, but it's always automatically part of a Spring Boot Starter web project. In the web environment, it's often about implementing RESTful web services that exchange JSON documents.

If a starter doesn't tie Jackson in, we supplement it in the POM:

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-json</artifactId>
</dependency>
```

The starter doesn't explicitly name Jackson.

3.9.3 Write a Java Object in JSON

Suppose we aim to create a web service endpoint that furnishes registration statistics. To construct these registration statistics, we require a JSON document resembling the following structure:

---

[76] The term "serialization" should be used only for the special Java SE format of `Object[Output|Input]Stream`.

```
{
  "data": [
    {
      "x": "2020-01",
      "y": 100
    },
    {
      "x": "2022-02",
      "y": 300
    },
    {
      "x": "2022-09",
      "y": 500
    }
  ]
}
```

What does the Java class look like so that Jackson can generate a JSON with this format or build a Java object from the JSON document? The class could look like this (records work as well):

```
public class Registrations {
  public static class Data {
    @JsonProperty( "x" ) public YearMonth yearMonth;
    @JsonProperty( "y" ) public int count;

    public Data() { }
    public Data( YearMonth yearMonth, int count ) {
      this.yearMonth = yearMonth;
      this.count = count;
    }
  }

  public List<Data> data;

  public Registrations() { }
  public Registrations( List<Data> data ) {
    this.data = data;
  }
}
```

The container class is called `Registrations`, where the class name doesn't matter in JSON mapping because the root object has no name—it would be different in XML. `Data` is a nested class with two attributes. Java SE conveniently declares the `YearMonth` data type. The variables could have been named `x` (instead of `yearMonth`) and `y` (instead of `count`), but the variables get “good” names, and `@JsonProperty` renames them for JSON mapping.

The `Registrations` class references a list of data objects, and the `List` becomes a JSON array. The constructors aren’t mandatory, but they are handy. The container classes can also be written with Java records, and Jackson can handle that. You are welcome to rewrite this as an exercise.

Perform JSON Mappings with ObjectMapper

The marshaling of JSON and Java objects is handled by Jackson’s `ObjectMapper` class. By default, Spring doesn’t build an `ObjectMapper`, but there is a `Jackson2ObjectMapperBuilder`,[77]which helps to build an `ObjectMapper` instance.

If an `ObjectMapper` should appear preconfigured as a Spring-managed bean in the context, the code could look like this:

```
@Configuration
public class ObjectMapperConfig {
  @Bean //  @Primary
  public ObjectMapper customJson( Jackson2ObjectMapperBuilder builder ) {
    return
      // new Jackson2ObjectMapperBuilder()
      builder.indentOutput( true )
        .serializationInclusion(JsonInclude.Include.NON_NULL)
      //.propertyNamingStrategy(PropertyNamingStrategies.UPPER_CAMEL_CASE)
        .build();
  }
}
```

The factory method falls back on an injected `Jackson2ObjectMapperBuilder` object to build the `ObjectMapper` instance, but there are two alternatives:

- Build your own builder instance with `new Jackson2ObjectMapperBuilder()`.
- Build an `ObjectMapper` directly with `new ObjectMapper()`.

The chosen variant builds on what Spring Boot has preconfigured in `Jackson2ObjectMapperBuilder`. The `Jackson2ObjectMapperBuilder` is configured via a Fluent API. For example, indentation can be turned on or statements can be made about how to handle `null`.

The factory method creates the component that can be injected elsewhere:

```
@Autowired ObjectMapper mapper;
```

---

[77] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/http/converter/json/Jackson2ObjectMapperBuilder.html*

`Registration` objects can be built and converted to a JSON string with `writeValueAsString(…)`. In code, for example, it looks like this:

```
Registrations registrations = new Registrations(
    Arrays.asList(
        new Registrations.Data( YearMonth.of( 2020, 1 ), 100 ),
        new Registrations.Data( YearMonth.of( 2022, 2 ), 300 ),
        new Registrations.Data( YearMonth.of( 2022, 9 ), 500 )
    ) );
System.out.println( mapper.writeValueAsString( registrations ) );
```

3.9.4 Testing JSON Mappings: @JsonTest

With the many annotations and marshaling adjustments of the JSON libraries, a JSON document may later look quite different from the class. A test case must show that the mapping is correct. It's important to test in two directions:

- Whether the Java document can be generated from the JSON document in a correct way
- Whether the correct JSON document is generated from the Java object.

Spring Boot provides a variety of options for testing JSON Java object mappings. First, let's look at the `@JsonTest` annotation for a test slice; then only a fraction of Spring components are built:

- The auto-configuration initializes a JSON library such as Jackson. Spring detects what is in the classpath as usual.
- Spring Boot builds a special helper class, in the case of Jackson, it's `JacksonTester`. For the other JSON libraries, there are other corresponding test classes: `GsonTester` and `JsonbTester`. We'll stick to Jackson in this book. The `JacksonTester` can be used later to convert the objects to JSON and vice versa.
- Spring Boot supports classes annotated with `@JsonComponent` to program mappings individually.

Further components aren't recognized and registered because they aren't necessary for a test of the mapping. If objects are needed, nevertheless, `@Import` helps to include them.

JacksonTester

We will begin by outlining the implementation of the test class.

```
@JsonTest
class JsonRegistrationsTest {

  @Autowired
  private JacksonTester<Registrations> jacksonTester;

  @Test
  void marshal_registrations() throws IOException {

    Registrations registrations = new Registrations(
        Arrays.asList(
            new Registrations.Data( YearMonth.of( 2020, 1 ), 100 ),
            new Registrations.Data( YearMonth.of( 2022, 2 ), 300 ),
            new Registrations.Data( YearMonth.of( 2022, 9 ), 500 )
        ) );

    JsonContent<Registrations> result = jacksonTester.write( registrations );

    // String json = result.getJson();

    JsonContentAssert jsonContentAssert = Assertions.assertThat( result );
    // …
  }
}
```

Listing 3.29 JsonRegistrationsTest.java

The class is no longer `@SpringBootTest`, but `@JsonTest` for the test slice. An object of the type `JacksonTester` is built as a Spring-managed bean and injected into the test class. `JacksonTester` always takes care of special container objects—in our case, they are `Registrations` objects—and therefore the type is built with a generic argument.

The class will later contain two test cases: one for mapping an object to JSON, and then, vice versa, another from JSON to the Java object. `marshal_registrations()` takes care of the first part.

Testing the "Java to JSON" Mapping with JsonContentAssert and JSON Paths

The test case builds and configures a `Registrations` object in the first step. To convert the Java object to JSON, the injected `JacksonTester` calls the `write(…)` method. The result is a `JsonContent` object,[78] which is parameterized as a generic type with `Registrations`.

The `JsonContent` object returns a JSON string with `getJson()` that a test could examine, but there's another way: `Assertions.assertThat(…)` can accept a `JsonContent`, and then the structure of the JSON document can be checked via a Fluent API. This is done by using JSON path expressions.

---

[78] *https://docs.spring.io/spring-boot/docs/current/api/org/springframework/boot/test/json/JsonContent.html*

JSON paths are related to XPath in XML. A special engine maps a path specification to a result. As an example, consider the JSON string `{"data":[{"x":"2020-01","y":100},{"x":"2022-02","y":300},…]}`.

JSON Path	Result
`$.data[1]`	`[{"x": "2022-02","y": 300}]`
`$.data[0,2]`	`[{"x": "2020-01","y": 100}, {"x": "2022-09", "y": 500}]`
`$.data[1].x`	`["2022-02"]`
`$.data[*].x`, `$.data..x`, or `$..x`	`["2020-01"`, `"2022-02"`, or `"2022-09"]`
`$.data.length`	`[3]`

Table 3.2 Examples of JSON Paths

Such path specifications can be used for a test case. Here we come back to `JsonContentAssert`.[79] The class has more than 70 methods—not counting the methods of superclass `AbstractAssert`. As with AssertJ, the Fluent API is used to express the requirements for the data. The peculiarity is that JSON paths are the focus of our queries.

The following listing is included in the test method.

```
// {"data":[{"x":"2020-01","y":100},{"x":"2022-02","y":300},…]}

assertThat( result )
    .hasJsonPathArrayValue( "$.data" );

assertThat( result )
    .extractingJsonPathStringValue( "$.data[1].x" ).isEqualTo( "2022-02" );

assertThat( result )
    .extractingJsonPathNumberValue( "$.data[1].y" ).isEqualTo( 300 );

assertThat( result )
    .isEqualToJson( "{\"data\":[{\"x\":\"2020-01\",\"y\":100}," +
                    "{\"x\":\"2022-02\",\"y\":300}," +
                    "{\"x\":\"2022-09\",\"y\":500}]}" );
```

Listing 3.30 JsonRegistrationsTest.java: marshal_registrations

Method `hasJsonPathArrayValue(…)` checks if the array `data` exists and has values. `extractingJsonPathStringValue(…)` retrieves the second element of the array with

---

[79] *https://docs.spring.io/spring-boot/docs/current/api/org/springframework/boot/test/json/JsonContentAssert.html*

`data[1].x` and checks if property `x` is equal to `2022-02`. Property `y` is a numeric value that `extractingJsonPathNumberValue(…)` extracts. With `isEqualToJson(…)`, the whole JSON can be tested.

The mapping we saw was from a Java object to a JSON object; JSON paths helped with the matching. The Spring Boot Starter Test brings the dependencies for the JSON paths.

Testing the "JSON to Java Object" Mapping

The other direction, from JSON to Java, is much easier:

```
@Test
void unmarshal_registrations() throws IOException {

  String json = "{\"data\":[{\"x\":\"2020-01\",\"y\":100},"
            + "{\"x\":\"2022-02\",\"y\":300},{\"x\":\"2022-09\",\"y\":500}]}";

  Registrations result = marshaller.parse( json ).getObject();

  assertThat( result.data ).hasSize( 3 );
  assertThat( result.data ).element( 0 )
      .extracting( data -> data.count ).isEqualTo( 100 );
}
```

The `JacksonTester` declares a `parse(…)` method that returns a Java object, in our case, a `Registrations` object. The familiar `assertThat(…)` calls are used for this purpose. For example, array `result.data` can be looked at and asked if it contains 3 elements and if the count field of the first element in the list is 100.

3.9.5 Mapping with @JsonComponent

Standard marshaling is tightly coupled with the JSON library; for example, if Jackson were replaced with Gson, all the annotations would no longer be recognized, and the mapping would be broken. In addition, the mapping might be so complex that it can't be described declaratively by annotations.

Spring Boot is capable of autonomously performing the mapping process, which is facilitated by `@JsonComponent` classes. To conclude, let's consider an example of how a secondary `Registrations2` class may be mapped.

```
public class Registrations2 {
  public static class Data {
    public int year, month;   // -> year + "-" + ("0"?) + month
    public int count;

    public Data() { }
    public Data( int year, int month, int count ) {
      this.year = year;
      this.month = month;
      this.count = count;
    }
  }

  public List<Data> data;

  public Registrations2() { }
  public Registrations2( List<Data> data ) {
    this.data = data;
  }
}
```

Listing 3.31 Registrations2.java

From the code, you can see that `Registrations2` doesn't reference a `YearMonth` object, but instead references two instance variables for the year and the month. If the mapping should be as before, the year must be followed by a minus sign and then the month. This means that the mapping needs some logic.

The `@JsonComponent` classes have the following structure:

```
@JsonComponent
public class RegistrationSerializer {

  public static class RegistrationsJsonSerializer
      extends JsonSerializer<Registrations2> {
    @Override
    public void serialize( Registrations2 registrations,
                           JsonGenerator out,
                           SerializerProvider serializerProvider )
        throws IOException {
      …
    }
  }

  public static class RegistrationsJsonDeserializer
      extends JsonDeserializer<Registrations2> {
    @Override
    public Registrations2 deserialize( JsonParser __,
                                       DeserializationContext ___ ) {
      throw new UnsupportedOperationException();
    }
  }
}
```

Annotation type `@JsonComponent`[80] is itself annotated with `@Component`, similar to `@ShellComponent` and other special components.

There are several notations for an `@JsonComponent` class. The implementation shown chooses two nested classes that extend `JsonSerializer`[81] and `JsonDeserializer`—neither of which are Spring data types. `JsonSerializer` maps the Java object to JSON in the `serialize(…)` method. Spring Boot passes the object to be mapped to the method and a `JsonGenerator` that the method can use to write the object; we'll look at the code shortly. `deserialize(…)` implements the opposite way: from a JSON object to the Java object. This isn't implemented, so the method throws an `UnsupportedOperationException`.

Next, you'll see an example of how the `serialize(…)` method can be implemented.

Implementation of a serialize(…) Method

This is how an implementation could look:

```
public void serialize( Registrations2 registrations,
                       JsonGenerator out,
                       SerializerProvider __ ) throws IOException {

  out.writeStartObject();
  out.writeArrayFieldStart( "data" );
  for ( Registrations2.Data data : registrations.data ) {
    out.writeStartObject();
    out.writeStringField( "x",
                          String.format("%d-%02d", data.year, data.month) );
    out.writeNumberField( "y", data.count );
    out.writeEndObject();
  }
  out.writeEndArray();
  out.writeEndObject();
}
```

Spring Boot hands the object to be mapped (`Registrations2`) to the `serialize(…)` method and the `JsonGenerator` to write to, as well as a `SerializerProvider`, but the program doesn't need that.

`com.fasterxml.jackson.core.JsonGenerator` is related to `XMLStreamWriter` from the Java StAX API; different methods express that an object starts, an object ends, an array starts, an array ends, or JSON properties are written. Besides mapping the hierarchy, the only thing our program needs to do is to iterate over the `data` array and put the year and month together with a minus sign.

---

[80] *https://docs.spring.io/spring-boot/docs/current/api/org/springframework/boot/jackson/JsonComponent.html*

[81] *https://fasterxml.github.io/jackson-databind/javadoc/2.9/com/fasterxml/jackson/databind/JsonSerializer.html*

`@JsonComponent` becomes a Spring-managed bean and is used automatically when reading and writing. In the test case, we could have the `RegistrationSerializer` component injected, or we can observe the desired effects:

```
@Autowired
private JacksonTester<Registrations2> jacksonTester2;

@Test
public void marshal_registrations2() throws IOException {
  Registrations2 registrations = new Registrations2(
      Arrays.asList(
          new Registrations2.Data( 2020, 1, 100 ),
          new Registrations2.Data( 2022, 2, 300 ),
          new Registrations2.Data( 2022, 9, 500 )
      ) );

  JsonContent<Registrations2> result = jacksonTester2.write( registrations );
  System.out.println( result.getJson() );
  // ->
  // {"data":[{"x":"2020-01","y":100},{"x":"2022-02","y":300},…
}
```

The `JacksonTester` will be parameterized with `Registrations2`, and Spring will use the `@JsonComponent` class for reading and writing. In other words, there is no need to explicitly use the converter, but Spring Boot will use our implemented mappings when a `Registration2` object is written or read. For the application (or test), this mapping is transparent, occurring in the background.

This was the first of several slice tests. In the following chapters, we'll come back to other test slices, especially `@DataJpaTest` (see Chapter 7, Section 7.16.2) and `@WebMvcTest` (see Chapter 9, Section 9.15.2).

3.10 Scheduling *

One of the useful features of the Spring Framework is the ability to perform *scheduling* to ensure that certain methods are called at specific intervals. This feature can be very useful in situations where you need to perform a task repeatedly, such as generating statistics every hour, cleaning up temporary files at night, or performing backups once a day.

3.10.1 Scheduling Annotations and @EnableScheduling

The built-in scheduling is enabled via `@EnableScheduling`,[82] which configures a `ScheduledAnnotationBeanPostProcessor` that looks at certain annotations and calls the marked methods at repeated intervals. As usual, the enabler must be at any `@Configuration`:

```
@Configuration
@EnableScheduling
class SchedulingConf { }
```

Because `@SpringBootApplication` is a special `@Configuration`, `@EnableScheduling` would be allowed there as well.

3.10.2 @Scheduled

Methods that are to be called repeatedly are annotated with `@Scheduled`.[83] Annotation attributes determine in which frequency the method should be called. The basic structure is as follows:

```
@Component
class Repeater {

  @Scheduled( … )
  public void repeatMe() { }

}
```

Two things need to be considered:

- The method must not have a parameter list. Spring calls the method and passes nothing to the method; auto-wiring of parameters is also not possible.
- The method doesn't return anything because there's nowhere for the results to go.

Scheduled with fixedDelay or fixedRate

There are different annotation attributes to determine the repeat frequency for a method such as `repeatMe()`. With `fixedDelay`, Spring calls the method and then waits for the specified time unit (by default, this is milliseconds), which are specified with `fixedDelay`:

```
@Scheduled( fixedDelay = 1000 /* ms */ )
public void wait1SecondAfterCallThenRepeat() { }
```

---

[82] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/scheduling/annotation/EnableScheduling.html*

[83] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/scheduling/annotation/Scheduled.html*

Spring will wait for one second after the method execution and then call the method again. Whether the method finishes quickly or not doesn't matter: the interval between method calls is always one second (ideal). The situation is different with the next option.

With `fixedRate`, Spring calls the method periodically like a metronome:

```
@Scheduled( fixedRate = 1000 /* ms */ )
public void calledEverySecondMaybeParallel() { }
```

In this example, Spring will call the method every second. The consequence could be that the method runs several times at the same time—for example, if the method takes two seconds. Undeterred, Spring will call the method again every second.

Using `fixedDelay` and `fixedRate` like this, the processing starts immediately. But an `initialDelay` can defer the initial call:

```
@Scheduled( initialDelay = 1000 /* ms */, fixedRate = 5000 /* ms */ )
public void wait1SecondThenRepeadEvery5Seconds() { }
```

In the example, an initial delay of one second is set.

The time units can also be changed via the `timeUnit` attribute,[84] and the unit is of type `java.util.concurrent.TimeUnit`.

The next possibility to specify the repetition frequency are cron expressions.

Cron Expressions

CronJobs complete tasks under Unix systems at fixed times in the background. In the core there are six segments for a setting or repetition, namely in the accuracy: seconds, minutes, hours, day of the month, month, and day of the week.

Schematically, it looks like this:

```
┌───────────── Seconds (0-59)
│ ┌───────────── Minutes (0-59)
│ │ ┌───────────── Hours (0-23)
│ │ │ ┌───────────── Day of the month (1-31)
│ │ │ │ ┌───────────── Month (1-12)
│ │ │ │ │ ┌───────────── Day of the week (0-7)
│ │ │ │ │ │
│ │ │ │ │ │
* * * * * *
```

---

[84] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/scheduling/annotation/Scheduled.html#timeUnit*

Here are three examples:

- `* * * *`: every minute
- `0 * * *`: every hour
- `*/10 * * *`: every 10 seconds

The CronJob syntax is complex, but websites such as *https://crontab-generator.org/* or *https://crontab.guru/* help with "piecing together" the expression.

With the `@Scheduled` annotation, a cron expression can be set using the `cron` annotation attribute. Here's an example:

```
// second, minute, hour, day-of-month, month, day(s) of week
@Scheduled( cron = "*/5 * * * * MON-FRI" )
public void cronEveryFiveMinutesWorkingDays() { }
```

The cron expression causes the method to be called every five minutes from Monday to Friday. `zone` can also be used to specify a time zone.

Because cron expressions aren't that easy to read, Spring has introduced special macros that can express typical repetitions:

```
@Scheduled( cron = "@hourly" )// @daily/@midnight,@weekly,@monthly,@yearly
public void cronMacro() { }
```

Details and examples are available at *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/scheduling/support/CronExpression.html*.

@Scheduled Annotation Type

The `@Scheduled` annotation type is declared as follows:

```
@Target({ElementType.METHOD, ElementType.ANNOTATION_TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Repeatable(Schedules.class)
public @interface Scheduled {
        String CRON_DISABLED = ScheduledTaskRegistrar.CRON_DISABLED;
        String cron() default "";
        String zone() default "";

        long fixedDelay() default -1;
        String fixedDelayString() default "";

        long fixedRate() default -1;
        String fixedRateString() default "";
        long initialDelay() default -1;
        String initialDelayString() default "";
}
```

We've already talked about `fixedDelay`, `fixedRate`, `initialDelay`, `cron`, and `zone`. Interestingly, for `*delay` and `*rate`, there are two ways to specify them: once via a number of type `long` and then via a SpEL expression as a `String`. SpEL can address external properties in a configuration file, for example, and this is useful. For example, you can control that in one case there are no repetitions, but in another case, there are.

3.10.3 Disadvantages of @Scheduled and Alternatives

The `@Scheduled` annotation sounds helpful, but its capabilities are comparatively low, and there are drawbacks as well:

1. Once Spring has started a background thread for execution, the timer runs forever. While `initialDelay` can delay execution, there is no (easy) way to specify the number of repetitions or to stop the timer later.
2. CronJobs can be bound to fixed times. This is a problem if, for example, a method should be called at 12 o'clock at night to delete the temporary folder, but one second before 12, the server crashes, and shortly after 12 o'clock, the system is active again: the system misses the time of the CronJob due to this incident, and consequently the operation doesn't run.

Note: Outlook

The scheduling feature in the Spring Framework has been a topic of discussion among developers. Some have questioned why the scheduling capability in Spring seems limited compared to other frameworks. One possible explanation is that Spring is designed as an integration framework, meaning it provides a unified platform for integrating various technologies, rather than trying to offer a comprehensive solution for every feature in its own library. Scheduling requires more than just a simple annotation; it involves state and transaction management, among other things. As a result, many developers turn to third-party libraries such as *Quartz* (*www.quartz-scheduler.org/*) for more robust scheduling functionality. Quartz is an open-source library for scheduling jobs in Java, which is widely used in enterprise applications and is backed by Software AG.

3.11 Types from org.springframework.*.[lang|util]

In the following, we take a closer look at selected data types of the packages `org.springframework.lang`, `org.springframework.util`, and `org.springframework.data.util`.

3.11.1 org.springframework.lang.*Null* Annotations

`NullPointerException`s have been a vexing problem until now. Fortunately, code checkers can now successfully check whether there is a deference to a `null` expression. However, a code checker has problems detecting whether methods from foreign libraries return `null` or never return `null`. For this reason, various annotations have been declared recently, as listed here, that can express the following: “The method could return `null`” or “All methods of the class never return `null`.”

- `@NonNull`
- `@NonNullApi`
- `@NonNullFields`
- `@Nullable`

Development environments such as IntelliJ or Eclipse can evaluate these annotations and alert us via warnings that we’re running into a potential `NullPointerException`. In addition, there are external tools, the code checkers, which can also perform a program flow analysis.

The Spring Framework itself uses these annotations in many places, and if you move the mouse pointer over methods in IntelliJ, this is also displayed. We can also use the annotations and benefit from the precise `null` checking of the compiler.

Note

Annotation type `@NonNull` is annotated with `@javax.annotation.Nonnull` from JSR 305. This means that code checkers that check for `@Nonnull` will also find the `@NonNull` type from Spring:

```
package org.springframework.lang;
import …
import javax.annotation.Nonnull;
@Target({ElementType.METHOD, ElementType.PARAMETER, ElementType.FIELD})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Nonnull
@TypeQualifierNickname
public @interface NonNull { }
```

The `org.springframework.lang`[85] package is relatively small and currently only contains the four annotations: `@NonNull`, `@NonNullApi`, `@NonNullFields`, and `@Nullable`.

---

[85] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/lang/package-summary.html*

3.11.2 Package org.springframework.util

In package `org.springframework.util`,[86] there are more types; here's a small selection:

- **`MultiValueMap<K,V>` (interface)**
  This type associates a key with a list of values. There is no such thing in Java SE. There is the datatype `Set`, which contains a value only once, and a `Map`, which can also associate a key with exactly one single value (the associated value can represent a collection, of course). If you want to associate a key with a collection of values, you either have to program this yourself, or you can use the `MultiValueMap` implementation `LinkedMultiValueMap`.[87]
- **`Assert`**
  This tpe helps to check method parameters for validity. Usually, methods always look the same: First the passed arguments are checked (e.g., if the value is in the valid ranges), and an exception reports errors. If everything is OK, then it goes on to the actual functionality of the method or constructor. These checks are often programmed with `if` statements and, in case of error, the code `throw new IllegalArgumentException(…)`. This is exactly what the `Assert` class does for us. It provides many static test methods, and if the condition is false, an exception is thrown automatically. Examples are `Assert.notNull(…)` and `Assert.isTrue(…)`. The `Assert` class reduces the tests of the arguments to one-liners.
- **`StreamUtils`**
  This type copies between IO streams. Example methods are `copyRange(InputStream in, OutputStream out, long start, long end)` and `copyToString(InputStream in, Charset charset)`.
- **`StringUtils`**
  The class `java.lang.String` contains everything necessary, but there is a need for additional methods. A few additional utility methods are provided by Spring's `StringUtils` class. It doesn't contain as many methods as *Apache Commons Lang* (*https://commons.apache.org/proper/commons-lang/*), which provides about 200 additional string methods, but it does give useful additions, for example:
  - `Set<String> commaDelimitedListToSet(String str)`
  - `int countOccurrencesOf(String str, String sub)`
  - `String deleteAny(String inString, String charsToDelete)`
  - `String getFilenameExtension(String path), boolean hasText(CharSequence str)`
  - `String[] removeDuplicateStrings(String[] array)`
  - `String stripFilenameExtension(String path)`

---

[86] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/util/package-summary.html*

[87] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/util/LinkedMultiValueMap.html*

Note: Consideration

A very basic question is whether these types should be used at all because they bind our own code to the Spring Framework. If our program works with the Spring Framework anyway, then why not? These utility libraries are open to us. In addition, you don't want to reprogram the functionality. One solution would be to go to other open-source libraries, for example, *Apache Commons* (*https://commons.apache.org*) or *Google Guava* (*https://github.com/google/guava*). But this won't solve the problem because we only have a dependency on another library. This dependency wouldn't be as big as to the Spring Framework, but still, it exists, and it would be another dependency that we have to keep an eye on in terms of security issues. If you don't want to write everything yourself, you can't avoid a dependency. With Spring, they also know that native compilation is the long-term goal for all the building blocks of the framework.

3.11.3 Package org.springframework.data.util

Besides what the Spring Framework itself declares in terms of utility classes, there are also interesting data types in the other Spring projects, for example, in the *Spring Data Commons project*. There, in the `org.springframework.data.util`[88] package, you'll find interesting types such as `Optionals`, `Streamable`, `StreamUtils`, `Lazy`, and so on.

For those who don't use a *Spring Data* module but are still interested in the utility classes, the following must be included in the POM:

```
<dependency>
<groupId>org.springframework.data</groupId>
<artifactId>spring-data-commons</artifactId>
</dependency>
```

Optionals

`Optionals` is, as the name suggests, about static helper methods around the data type `java.util.Optional`:

- `<S,T> Optional<T> firstNonEmpty(Iterable<S> source, Function<S,Optional<T>> function)`
- `<S,T> T firstNonEmpty(Iterable<S> source, Function<S,T> function, T defaultValue)`
- `<T> Optional<T> firstNonEmpty(Iterable<Supplier<Optional<T>>> suppliers)`
- `<T> Optional<T> firstNonEmpty(Supplier<Optional<T>>... suppliers)`
- `<T,S> void ifAllPresent(Optional<T> left, Optional<S> right, BiConsumer<T,S> consumer)`

---

[88] *https://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/util/package-summary.html*

- `<T> void ifPresentOrElse(Optional<T> optional, Consumer<? super T> consumer, Runnable runnable)`
- `boolean isAnyPresent(Optional<?>... optionals)`
- `<T,S,R> Optional<R> mapIfAllPresent(Optional<T> left, Optional<S> right, BiFunction<T,S,R> function)`
- `<T> Optional<T> next(Iterator<T> iterator)`
- `<T> Stream<T> toStream(Optional<? extends T>... optionals)`
- `<T,S> Optional<Pair<T,S>> withBoth(Optional<T> left, Optional<S> right)`

Streamable

The idea behind `Streamable`[89] is to allow an alternative data type for returns. If the return is of type `Iterable`, it's of little value for the receiver because there isn't much that can be done with an `Iterable`: you can ask for an `Iterator` or iterate over the elements. `Iterator` is also not an interesting return type. If a method returns a list, that's more interesting, but often you'll ask for a stream from the list, so you can use the nice `filter(…)/map(…)/reduce(…)` methods. This is where `Streamable` comes into play.

The `Streamable` data type is frequently encountered in the Spring Data applications. See Figure 3.25 for an overview of the type relationships and methods associated with `Streamable`.

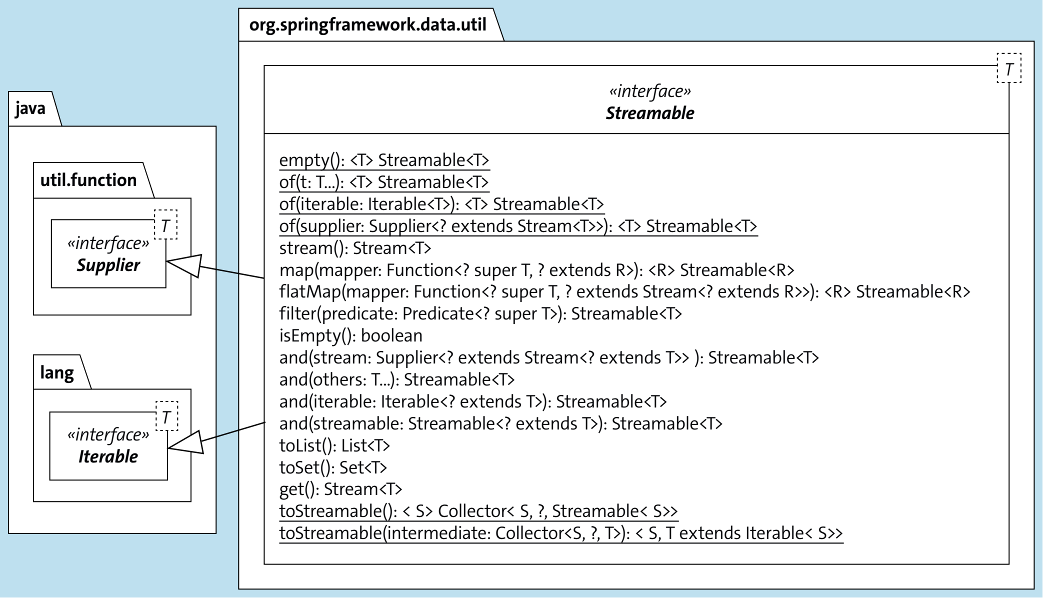

org.springframework.data.util
java
util.function
T
«interface»
Supplier
lang
T
«interface»
Iterable
T
«interface»
Streamable
empty(): <T> Streamable<T>
of(t: T...): <T> Streamable<T>
of(iterable: Iterable<T>): <T> Streamable<T>
of(supplier: Supplier<? extends Stream<T>>): <T> Streamable<T>
stream(): Stream<T>
map(mapper: Function<? super T, ? extends R>): <R> Streamable<R>
flatMap(mapper: Function<? super T, ? extends Stream<? extends R>>): <R> Streamable<R>
filter(predicate: Predicate<? super T>): Streamable<T>
isEmpty(): boolean
and(stream: Supplier<? extends Stream<? extends T>>): Streamable<T>
and(others: T...): Streamable<T>
and(iterable: Iterable<? extends T>): Streamable<T>
and(streamable: Streamable<? extends T>): Streamable<T>
toList(): List<T>
toSet(): Set<T>
get(): Stream<T>
toStreamable(): < S> Collector< S, ?, Streamable< S>>
toStreamable(intermediate: Collector<S, ?, T>): < S, T extends Iterable< S>>

Figure 3.25 UML Diagram Showing the Type Relationships of "Streamable"

---

[89] *https://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/util/Streamable.html*

The type hierarchy isn't exciting: `Streamable` is an extension of `Iterable`, and `Streamable` is also an extension of `Supplier`. That is, whenever `Iterable` is needed, we could also put something in it that is of type `Streamable`.

Many of the methods are known from the Java Streaming API. Several new `and(…)` methods append other elements to the current `Streamable`, making the stream larger.

Consider the following example that uses the `Streamable` data type:

```
Iterable<Path> directories = FileSystems.getDefault().getRootDirectories();
Set<String> strings =
    Streamable.of( directories )
              .map( String::valueOf )
              .and( "cloud" )
              .toSet();
System.out.println( strings ); // [cloud, P:\, C:\]
```

The starting point is `Iterable`. Because the data type is weak, we "upgrade" it to `Streamable`. The `Streamable` interface provides a set of static `of(…)` methods for creating `Streamable` objects.

The methods are known from the Stream API of Java SE. Thus, in the example, the `map(…)` method converts these paths into a string representation; at the end, there is again a `Streamable`. The convenient `and(…)` method appends an element to `Streamable`. Finally, the program converts the data into a `Set`.

Spring Data repositories can return `Streamable` objects, as we'll see later in Chapter 7, Section 7.8.3.

StreamUtils

The `StreamUtils` class[90] contains methods to build `Stream` or `Collector` instances. This isn't really anything that inherently belongs in the Spring Data package. The static methods are as follows:

- `<T> Stream<T> createStreamFromIterator(CloseableIterator<T> iterator)`
- `<T> Stream<T> createStreamFromIterator(Iterator<T> iterator)`
- `<T> Stream<T> fromNullable(T source)`
- `<T,K,V> Collector<T,MultiValueMap<K,V>,MultiValueMap<K,V>> toMultiMap(Function<T,K> keyFunction, Function<T,V> valueFunction)`
- `<T> Collector<T,?,List<T>> toUnmodifiableList()`
- `<T> Collector<T,?,Set<T>> toUnmodifiableSet()`
- `<L,R,T> Stream<T> (Stream<L> left, Stream<R> right, BiFunction<L,R,T> combiner)`

---

[90] *https://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/util/StreamUtils.html*

The methods are self-explanatory, although the `zip(…)` method is a bit special: it combines two streams, and, like a zipper, both elements of the stream are combined into a new stream via a `BiFunction`.

Lazy

The Spring Data Commons family declares another data type, called `Lazy`,[91] which can be used beautifully outside of Spring Data. The data type is distantly related to `Optional`. However, when building an `Optional` object, the value must already be initialized and supplied. Of course, the `Optional` can also be empty; we can ignore this special case because an empty `Optional` is essentially for free.

There are situations where the value within an `Optional` object may be expensive to construct, yet the value might not be used at all. This is where the `Lazy` class can be useful.

`Lazy` doesn't have constructors, but `empty()` and two `of(…)` methods. Passing own values directly makes the `Lazy` optional, but more interesting is to pass a `Supplier`. Only when the `Lazy` container is asked for its value, the `get(…)` method from the `Supplier` is called. The `Supplier` only has to supply the value once because the `Lazy` object stores the value for later accesses. The methods of `Lazy` are self-explanatory and known from `Optional`.

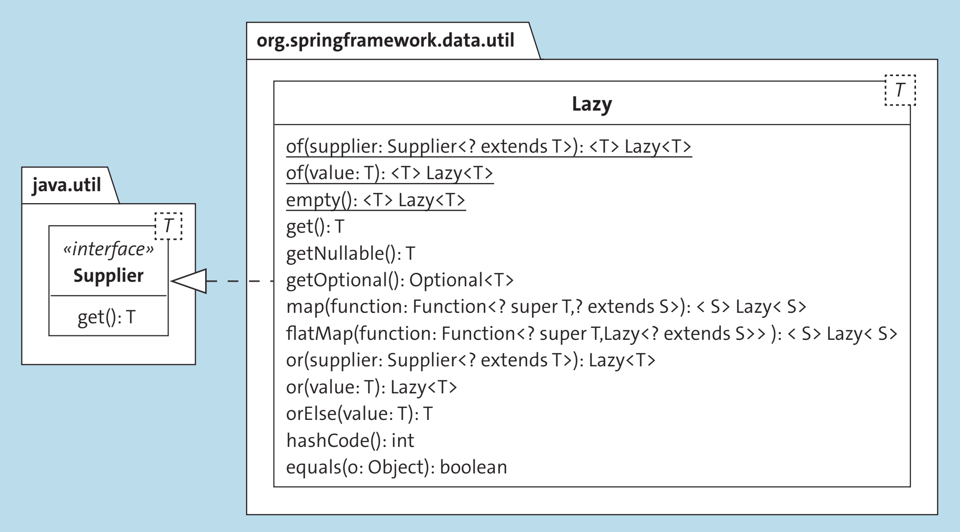

org.springframework.data.util
Lazy
T
of(supplier: Supplier<? extends T>): <T> Lazy<T>
of(value: T): <T> Lazy<T>
empty(): <T> Lazy<T>
get(): T
getNullable(): T
getOptional(): Optional<T>
map(function: Function<? super T,? extends S>): < S> Lazy< S>
flatMap(function: Function<? super T,Lazy<? extends S>>): < S> Lazy< S>
or(supplier: Supplier<? extends T>): Lazy<T>
or(value: T): Lazy<T>
orElse(value: T): T
hashCode(): int
equals(o: Object): boolean
java.util
T
«interface»
Supplier
get(): T

Figure 3.26 Methods and Type Relationship of Lazy

---

[91] *https://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/util/Lazy.html*

Pair

Sun Microsystems at the time—like Oracle today—always resisted building a data type for a pair. The Spring team didn't have these concerns and therefore declared a class `Pair`, although its use is rather discouraged. If small data structures are needed, a record can be used well, and then even primitive data types are no problem; with `Pair`, only two generic type variables are ever used.

3.12 Summary

The Spring Framework is a vast and complex framework due to mixing the container implementation with client code. There is no clear separation between the internal workings of the framework and our implementation, which can make it challenging to understand and work with. However, there are resources available to help developers navigate the complexity of Spring. For instance, curious individuals can run Spring programs in a debugger and browse the source code, which is available on GitHub *https://github.com/spring-projects/spring-framework*. Additionally, Spring documentation and community resources are readily available to provide guidance on how to work with the framework effectively.

In this chapter, I've omitted a presentation of aspect-oriented programming. If you like, you can read these details online in the reference documentation at *https://docs.spring.io/spring-framework/docs/current/reference/html/core.html#aop* and *https://docs.spring.io/spring-framework/docs/current/reference/html/core.html#aop-api*.

Chapter 4

Selected Proxies

Injection plays an important role in Spring applications. With the help of injection, objects are brought to the places where they are needed. However, the client may not get a reference to the desired object itself; instead, Spring sets up a proxy. A proxy can be thought of as a ring that wraps around an object, intercepts all method calls, and forwards them if necessary.

On the way to the target object in the core, there could be various checks, which could also go so far as to prevent access.

4.1 Proxy Pattern

What can be vividly explained using rings, we want to translate into the world of software engineering with a sequence diagram. The ring illustrates the *proxy pattern*, which is a well-known design pattern.

In the core shown in Figure 4.1 is the *target object*. This target object has an interface, let's call it `I`, with operations. The *proxy object* (aka substitute) wraps around this target object. It has the same interface `I` to the outside as the target object, which is also called the *subject*. If a client has a reference to “something of type `I`,” the client doesn’t know whether it’s talking to the proxy or to the target object.

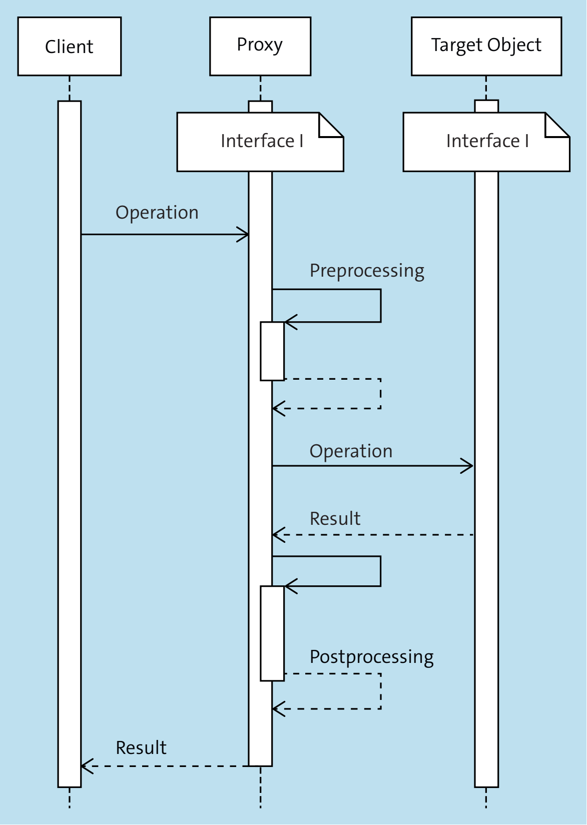

Client
Proxy
Target Object
Interface I
Interface I
Operation
Preprocessing
Operation
Result
Postprocessing
Result

Figure 4.1 Sequence Diagram of the Proxy Pattern

A proxy doesn’t sound exciting, but it can do two useful things:

- Before the proxy goes to the target object with the operation, it can perform preprocessing. That is, the proxy can call internal methods and modify the parameters. Then, the proxy can go to the target object or block the call.
- When the target object returns the result to the proxy, the proxy can perform postprocessing. For example, it can cache the data. Later, the proxy returns the result to the client.

Note

With the corresponding `Class` object, you can find out whether you have a proxy object in front of you or not. The `toString()` method also responds differently. Spring includes in `org.springframework.aop.support.AopUtils` the helper method `isAopProxy(…)` for Spring proxies:

```
public static boolean isAopProxy(@Nullable Object object) {
  return
    (object instanceof SpringProxy &&
        (Proxy.isProxyClass(object.getClass()) ||
    object.getClass().getName().contains(
    ClassUtils.CGLIB_CLASS_SEPARATOR)));
}
```

When implementing `equals(…)`, it must be taken into account in places that comparisons are programmed with `instanceof` instead of `getClass(…)`.

4.1.1 Proxy Deployment in Spring

In general, a proxy is limited in its abilities and is incapable of modifying the behavior of the target operation. However, the preprocessing and post-processing capabilities of a proxy are frequently sufficient for a variety of use cases. Consequently, Spring employs proxies in numerous contexts. On several occasions, the code is injected with a proxy, which often goes unnoticed because the proxy has the same interface as the target object.

Following are a few examples of how Spring uses proxies:

- **Transaction management**
  Methods can initiate new transactions, and everything the method does to the database takes place in a transactional context. If there are exceptions in the method, the proxy catches those exceptions and starts a rollback; if there are no errors, then a commit follows at the end of the method.
- **Caching**
  If a client calls a method with a certain argument, a proxy can determine that the method has seen the argument before and can return the result computed earlier.
- **Validation**
  The purpose of validation is to ensure that methods are called only with valid values. A proxy can check that the arguments are in the correct value ranges and pass them to the target object only if they are valid.
- **Asynchronous calls**
  Normally, a method is called synchronously (blocking). A special proxy can start a background thread—or fall back to a thread of a thread pool—and then asynchronously process this operation.
- **Spring Retry project**
  Aborts can occur—especially with remote accesses; the Spring Retry project can use a proxy that starts another attempt if a target object's operation fails, until the call finally succeeds or you give up.

To better comprehend how Spring operates with proxies, let's translate the concept into Java source code.

Proxy Pattern in Code

The proxy pattern, programmed in code, has the following basic form:

```
class Subject {
  public String operation( String input ) { return input; }
}

class Proxy extends Subject {
  private final Subject subject;

  Proxy( Subject subject ) { this.subject = subject; }

  @Override public String operation( String input ) {

    // Preprocess the input

    String result = subject.operation( input );

    // Postprocess the result

    return result;
  }
}
```

The proxy must be connected to the target object. This is how the constructor stores the reference.

What is symbolic here in Java code is automatically generated at runtime in the Spring Framework.

4.1.2 Dynamically Generate Proxies

The Spring Framework can generate proxies at runtime; this is why we refer to this form of proxy as *dynamic proxies*. The Spring Framework uses two techniques for this purpose.

- The Java Development Kit (JDK) can build dynamic proxies; these are called *JDK dynamic proxies*. In this case, there is a limitation because Java SE can only realize proxies for interfaces—this isn't possible for classes.
- When Spring generates a proxy for classes, the framework makes use of the library *cglib* (short for *Byte Code Generation Library*).[1] This Byte Code Generation Library is relatively old and no longer maintained, so the Spring team has heavily patched the version to make it fit for the current Java versions.

---

[1] *https://github.com/cglib/cglib*

Rules for Target Objects

If proxies are to work well, then the classes must take some rules into account. If Spring builds proxy objects for classes, then a library creates bytecode for a subclass. It follows that the superclass (the subject) must not have a private constructor. This is clear because a subclass always calls the constructor of the superclass, and if the superclass doesn't have a visible constructor, the constructor can't be called. Therefore, the constructors in the subject must not be private.

If Spring implements a proxy through a subclass, as we did in our own proxy example with the types `Proxy` and `Subject` in the previous section, then the proxy must override the method from the subject. Therefore, the method must not be `final`; otherwise, it could not be overridden.

In principle, all methods for which a proxy is generated should be `public`. Spring relies on its own aspect-oriented programming (AOP) framework, which requires `public` methods. Although all of these restrictions can be circumvented, then the bytecode must be modified via AspectJ. We'll only use the "usual" way here.

Besides the preceding rules, there are a few limitations if the framework is creating proxy objects. There is one thing to watch out for: A proxy can only do its job if it sits between the client and the subject. If the subject calls methods among themselves in their own code, then it won't go through the proxy. This can quickly lead to a problem, for example, when refactoring.

To illustrate the issue at hand, let's consider the example of a proxy class named `TrimmingProxy`. When parameter variables are annotated with `@Trim`, the `TrimmingProxy` class removes any leading or trailing white spaces from these strings before passing them to the target object. Here is an example usage of the `TrimmingProxy`:

```
IntParser proxy = new TrimmingProxy( new IntParser() );
proxy.parseInt( " 12423\t " );
```

Now let's look at the code of the `IntParser` class with the method:

```
class IntParser {
  public int parseInt( String input ) {
    return parseInt( input, 10 );
  }
  public int parseInt( @Trim String input, int radix ) {
    return Integer.parseInt( input, radix );
  }
}
```

If the proxy pays attention to `@Trim`, it will only be activated when `parseInt(String, int)` is called *directly*; otherwise, it won't be activated. Our example calls `parseInt(String)`, and the parameter variable has no `@Trim`, so the proxy won't remove white space either. If `parseInt(String)` later calls `parseInt(String, int)`, it happens inside the target object, and the proxy isn't involved.

> **Note**
>
> In practice, this is a mistake that is often made, so you must always remember that a proxy always goes into the core "from the outside" and only then does its job.

Build Proxies Yourself with ProxyFactory *

Spring provides the class `ProxyFactory`[2] with which we can also build proxy objects. Here's an example: A proxy object is to be created for a `java.util.List`. On this particular list, `add(null)` is to be ignored:

```
ProxyFactory factory = new ProxyFactory( new ArrayList<>() );
MethodInterceptor methodInterceptor = invocation ->
  (invocation.getMethod().getName() == "add"
  && isNull( invocation.getArguments()[ 0 ] )) ? false
                                               : invocation.proceed();
factory.addAdvice( methodInterceptor );
```

In the constructor of `ProxyFactory`, we pass the subject, an `ArrayList`. The goal is that `ProxyFactory` generates a proxy, and we pass certain methods to the list in the background.

The proxy intercepts every method call. It's therefore elementary to configure how the proxy should behave on which method and arguments. A `MethodInterceptor`[3] is helping us; the functional interface is declared as follows:

```
public interface MethodInterceptor extends Interceptor {
  @Nullable
  Object invoke(@Nonnull MethodInvocation invocation) throws Throwable;
}
```

If a method is called on the proxy from the outside, it passes the call to the configured `MethodInterceptor` and calls the `invoke(…)` method. In other words, no matter which method is accessed from the outside, it will always map to an `invoke(…)` method. The `MethodInvocation` parameter is so important because the object contains information about which method was called with which arguments.

Our implementation first tests with `invocation.getMethod().getName() == "add"` if the method `add` was really called. Because the strings of the method names are internal, it's not a problem to use == for the comparison; however, this is rare. The subsequent check with `isNull( invocation.getArguments()[ 0 ] )` checks if `null` was passed to the `add(…)` method. There are two outputs for the comparisons:

---

[2] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/aop/framework/ProxyFactory.html*

[3] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/aopalliance/intercept/MethodInterceptor.html*

- `add(null)` has been called. In this case, we return a `boolean` value because this is also the return type of the `add(…)` method. Returning `false` expresses that no change was made to the data structure. We don't pass anything to the target objet. Although `invoke(…)` only returns `Object`, it's important to make sure that the return type matches that of the target object in the method.
- If `add(null)` was *not* called, so either another method or `add(…)` with another value than `null`, then the proxy goes to the target object with `invocation.proceed()`, and what the subject returns, we return at `invoke(…)`.

With this code, the `MethodInterceptor` is implemented. After it's passed into `addAdvice(…)`, we can obtain and use the proxy object from the `ProxyFactory`. Here's an example:

```
List list = (List) factory.getProxy();
list.add( "One" );
System.out.println( list );
list.add( null );
list.add( "Two" );
System.out.println( list );
```

The outputs are as follows:

```
[One]
[One, Two]
```

Having discussed home-coded proxies, let's now examine some standard proxies provided by the Spring Framework.

4.2 Caching

In this section, we'll delve into the concept of caching, how it can be implemented in Spring applications, and the benefits it provides. *Caching* is an essential technique used to speed up application performance, reduce database load, and enhance user experience. With Spring's robust caching support, you can easily incorporate caching into your applications and reap its benefits.

4.2.1 Optimization through Caching

Caching plays a crucial role in scenarios where data retrieval is resource-intensive, and a quick cache is available to temporarily store the object. Caching has various practical applications:

- The Domain Name System (DNS) cache is used by every PC and smartphone to recall the IP address of a specific host name.
- Companies make use of a web proxy and cache wherein the proxy caches the retrieved components when queries are sent out from the company network.
- When a web server is used, not all queries require file access because the server retains the file contents and only refreshes the cache entry when modifications are made to the file.
- Spring applications often store the contents of a database in the cache to minimize database access.

Caching Challenges

Caching is only worthwhile if the element being queried is relatively expensive to create or query.[4] Stored elements that are then invalidated are a particular challenge. For a purely idempotent operation, such as a sine function (something comes in and the same thing always comes out), caching is easy. But many entries eventually become invalid. For example, if a record is associated with an ID, then the record may change. Consequently, the cache content must be marked as *stale*.

A cache must also work like the human brain; that is, it must be able to forget. It's of little use if the cache occupies the entire main memory, and this leads to the `OutOfMemoryError` of the application. The cache must forget elements that haven't been used for a long time, just as probably all of us have forgotten the components of a flowering plant from the sixth grade.

4.2.2 Caching in Spring

In Spring, enabling caching is straightforward. The underlying mechanism involves a Spring proxy that associates method arguments with their respective return values. Additionally, it's easy to swap different caching implementations with minimal effort.

There are several approaches to implementing a caching mechanism in Spring. Usually, caching is realized declaratively, which means that the actual cache access isn't visible in your own code. There are two declarative approaches: *annotation-based* or *XML-based*. We'll omit the XML method because it's uncommon today, as is the whole container configuration.

There are two options for annotations:

---

[4] Back in the early days of Java, developers were concerned about the cost of object creation and often tried to reuse objects to save memory. However, this approach is generally not recommended today as the cost of object creation and cleanup has become relatively cheap. In fact, artificially keeping objects alive for too long can actually cause issues with memory management. This is due to the generation-based garbage collector moving these objects to the "old generation" where memory release isn't as fast as with fresh objects. As a result, it's important to carefully consider object lifecycle management to avoid any unintended consequences.

- Spring annotations such as `@Cacheable`, `@CacheEvict`, `@CachePut`, `@Caching`, and `@CacheConfig`
- JCache annotations such as `@CacheResult`, `@CachePut`, `@CacheRemove`, `@CacheRemoveAll`, `@CacheDefaults`, `@CacheKey`, and `@CacheValue`

Note

JCache is the standard from JSR 107 (*www.jcp.org/en/jsr/detail?id=107*).

Not only are the Spring and JCache annotations named differently, but the semantics are different in places; JCache also caches exceptions. Spring Cache can also be configured with a JCache implementation.

In the following example, the Spring annotations are used.

@EnableCaching

The actual work in the background is done by proxy objects. A proxy will later wrap itself around the objects and cache the result of the annotated method. The proxies aren't built automatically, but for this purpose, the annotation `@EnableCaching`[5] has to be present on any configuration class:

```
@EnableCaching
@Configuration
class CacheConfig { }
```

Our main class that was annotated with `SpringBootApplication` is a special `@Configuration` class, so it would work there too.

If methods are later called that are annotated with `@Cacheable`, `@CacheEvict`, and so on, a proxy will receive the calls.

4.2.3 Component with the @Cacheable Method

Consider the following brief example of an operation that should store its output in the cache:

```
@Component
class HotProfileToJsonConverter {
  private final Logger log = LoggerFactory.getLogger( getClass() );

  @Cacheable( "date4u.jsonhotprofiles" )
  // @Cacheable( cacheNames = "date4u.jsonhotprofiles" )
```

---

[5] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/cache/annotation/EnableCaching.html*

```
  public String hotAsJson( List<Long> ids ) {
    log.info( "Generating JSON for list {}", ids );
    return ids.stream().map( String::valueOf )
              .collect( Collectors.joining( ",", "[", "]" ) );
  }
}
```

The `hotAsJson(…)` method gets a list of IDs featuring the hottest unicorns and responds with a JavaScript Object Notation (JSON) string.

There are three key aspects to consider when dealing with caching:

- The annotation `@Cacheable`[6] is used on specific methods to associate the method's arguments with its corresponding return value. Essentially, a cache is an associative data store that maps a key to a value, where the key is the argument passed to the method. With each unique argument passed, the proxy retains the resulting return value in memory.
- A cache always needs a name; a hierarchical structure is useful to avoid conflicts with other applications. The name can be set via annotation attribute `cacheNames` or `value`. The name is important for two reasons:
  - Different caches can be used within one class.
  - Different classes can use the same cache.
- The methods that the proxy calls must be `public`. We discussed this in Section 4.1.2 under "Rules for Target Objects" subsection. In principle, other visibilities would be possible, but the proxy must fall back on these methods, and this works best if the method is `public`. Other visibilities are possible, but then the bytecode must be wrapped with AspectJ.

Now let's see how we can use the `HotProfileToJsonConverter`.

4.2.4 Use @Cacheable Proxy

The `HotProfileToJsonConverter` is a Spring-managed bean and can be injected:

```
@Autowired HotProfileToJsonConverter converter;
```

What Spring injects is *not* our own object! We get something that is of type `HotProfileToJsonConverter`, but it's the proxy. In the debugger (or via the `Class` object), this is easy to see:

```
log.info( converter.getClass().getName() );
// c.t.d.HotProfileToJsonConverter$$EnhancerBySpringCGLIB$$8d40339b
```

---

[6] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/cache/annotation/Cacheable.html*

The name of the `Class` object contains the string `EnhancerBySpringCGLIB`, and this is the typical identification of a proxy generated at runtime.

This proxy has the `hotAsJson(…)` method that we can use:

```
String json1 = converter.hotAsJson( Arrays.asList( 1L, 2L, 3L ) );
String json2 = converter.hotAsJson( Arrays.asList( 1L, 2L, 3L ) );
// assertThat( json1 ).isSameAs( json2 );
```

The resulting strings aren't just equivalent—they are also identical. This is a crucial point, which is attributed to the working of the proxy.

Even the first call to `hotAsJson(…)` is received by the proxy. The proxy maps the argument to a key (how to do this in a customized way is shown in Section 4.2.9) and internally requests the cache with the question, "Is there already an associated result for the list with values 1, 2, 3?" The answer is "no" in the first query. Therefore, the proxy calls the core, gets the string, and puts it in the cache. With the second call of `hotAsJson(…)`, it already looks different: even then, the proxy goes to the cache first, but it has an entry for the list of 1, 2, 3. In other words, the second call doesn't go to the core, but the proxy returns the cached value. Therefore, it appears in the log output only once:

```
Generating JSON for list [1, 2, 3]
```

If you were to work with a second list, the same thing would start all over again:

```
String json3 = converter.hotAsJson( Arrays.asList( 1L ) );
// assertThat( jspn1 ).isNotSameAs( json3 );
```

The cache hasn't seen the list with 1 yet, so it goes to the core and caches the result. Of course, the JSON strings of lists 1, 2, 3 and 1 are entirely different.

Cache Images

We'll examine caching using a concrete example in our Date4u application.

In `PhotoService`, there is a `download(…)` method with the following implementation:

```
public Optional<byte[]> download( String imageName ) {
  try {
    return Optional.of( fs.load( imageName + ".jpg" ) );
  }
  catch ( UncheckedIOException e ) {
    return Optional.empty();
  }
}
```

Listing 4.1 PhotoService.java Excerpt

An image name is passed to the `download(…)` method, and the return is `Optional` with a `byte` array. Internally, the method calls the file system.

If the `download(…)` method is called twice, the image could easily come out of the cache as a `byte` array. It can be implemented as follows:

1. In any configuration, `@EnableCaching` must be present. Because `@SpringBootApplication` is a special kind of `@Configuration`, it is valid to apply the `@EnableCaching` to the class with the `main` method:

   ```
   @SpringBootApplication
   @EnableCaching
    public class Date4uApplication {
     public static void main( String[] args ) {
       SpringApplication.run( Date4uApplication.class, args );
     }
    }
   ```

   Listing 4.2 Date4uApplication.java Extension

2. For the `PhotoService`, a name must be added to the `@Cacheable` annotation.

   ```
   @Cacheable( "date4u.filesystem.file" )
    public Optional<byte[]> download( String imageName )
   ```

   Listing 4.3 PhotoService.java Extension

The cache name is arbitrary, and `date4u.filesystem.file` works and is unique.

4.2.5 @Cacheable + Condition

In some situations, specific items ought not to be cached. This could be due to their size or because they aren't frequently accessed. Thus, caching can be associated with certain criteria, such as if an object possesses particular attributes, it should not be cached.

In Spring, the `condition` annotation attribute[7] controls whether an object should be cached. If the `condition` is true, the object is cached. In the code, it looks like this:

```
@Cacheable( cacheNames = "date4u.jsonhotprofiles",
            condition  = "true" )
```

This condition is written as a Spring Expression Language (SpEL) expression (refer to Chapter 2, Section 2.11). SpEL is powerful and allows access to the object graph and to all parameters. `condition = true` is a simple SpEL expression and exemplifies the notation. By the way, the assignment `condition = true` is the default.

---

[7] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/cache/annotation/Cacheable.html#condition()*

Suppose that caching with `hotAsJson(…)` is only beneficial for extensive lists, and small lists should not be cached. This idea can be conveyed as follows:

```
@Cacheable( cacheNames = "date4u.jsonhotprofiles",
            condition  = "#ids.size() > 10" )
public String hotAsJson( List<Long> ids )
```

The condition `#ids.size() > 10` contains an expression regarding whether the list contains more than 10 elements. The variable `ids` is the parameter name that can be accessed by the SpEL expression. The parameter type is `java.util.List`, and it has a `size()` method. If the number of list items exceeds 10, then the cache should be asked and also filled. This is obvious: cache large lists, not small lists. If the condition is false—that is, the list is small—then the call goes directly to the core and also again from the target object directly to the client without caching the object.

For the condition, the parameters are usually used, and further identifiers are predefined in SpEL:

- `#root.method`, `#root.target`, `#root.caches`
- `#root.methodName`, `#root.targetClass`
- `#root.args[0]`, `#p0`, `#a0`, `root.args[1]`, and so on, or simply `#parameter-name`

You can fall back on the reflection `Method` object or, alternatively, on the method name, `Class` object, or class name.

There are different ways of writing to access the parameters. The most readable is the variant that our example uses: with the number sign and the parameter name.

4.2.6 @Cacheable + Unless

In the Spring caching framework, a veto is a mechanism that allows the programmer to prevent certain method invocations from being cached. More specifically, it ensures that a previously uncached result won't be cached even if the method is invoked multiple times with the same arguments.

This can be particularly useful in scenarios where the method may return `null` or empty results, as caching such outcomes could result in wasted memory and reduced performance. By setting a veto, the programmer can instruct the caching system to skip the caching process for specific method outcomes.

In Spring caching, the veto is expressed with the annotation attribute `unless`:[8]

```
@Cacheable( cacheNames = "date4u.jsonhotprofiles",
            unless     = "false" )
```

---

[8] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/cache/annotation/Cacheable.html#unless()*

The SpEL expression here is `false`, which means there is *no* veto, so the result goes into the cache.

`unless` set to `false` is only an example for clarification. In practice, a SpEL expression is used, and this can fall back on the known information, the method name, the parameter list, and so on. An additional variable, `result`, provides access to the result from the core in the SpEL expression. This can be used to decide whether the object goes into the cache or not.

Let's move on to something more practical with a couple of examples:

Example 1: A JSON result under 100 characters should not be cached:

```
@Cacheable( cacheNames = "date4u.jsonhotprofiles",
            unless     = "#result.length() < 100" )
```

If the `result`—it's a `String` here—has less than 100 characters, then it's a veto, and the string should not be cached. `unless` controls in this way that small strings aren't cached and that caching is only worthwhile if the strings become larger.

Example 2: If a method returns a collection, but the result is `null` or the collection is empty, this should be a veto. The SpEL expression could look like this:

```
#result == null || #result.size() == 0
```

When the veto is true, the collection is prevented from being stored in the cache. On the other hand, if there is no veto, and the collection isn't `null` or empty, it will be stored in the cache.

Remarks

If an exception occurs in the core that the cache proxy wraps, those exceptions aren't cached. That is, there is no association between a value and the exception that it doesn't work with the value.

If a method returns an `Optional`, the result doesn't contain the `Optional`, but contains either `null` (if the `Optional isEmpty()`) or the value from the `Optional`.

4.2.7 @CachePut

If a method is annotated with `@CachePut`,[9] the cache can be written directly. This is convenient when the cache is to be filled initially because, if the elements aren't present, the cache doesn't have to be queried first. `@CachePut` is also useful when elements in the cache are to be updated and overwritten, such as for outdated values. The underlying method in the core is thus always called, and the value is determined and stored.

---

[9] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/cache/annotation/CachePut.html*

`@CachePut` and `@Cacheable` have similar annotation attributes. But using both annotations together doesn't make sense, and we use either `@CachePut` or `@Cacheable`.

4.2.8 @CacheEvict

Effective cache management is essential to prevent it from becoming too large and impacting system performance. There are two common strategies for managing a cache's elements:

- **Automatic expiration**
  This strategy involves setting conditions that trigger the cache to remove elements on its own. For example, a cache may have a *time-to-live* (TTL) value that specifies the maximum amount of time an element can remain in the cache before it's expired and removed automatically. Another example is setting a maximum cache size, where the oldest elements are removed when the cache reaches its limit.
- **Manual eviction**
  In this strategy, the programmer manually deletes specific elements or clears the entire cache. This can be useful in scenarios where the cached elements are no longer valid or when memory needs to be reclaimed.

Choosing the appropriate strategy will depend on the specific use case and requirements of the application. Automatic expiration may be suitable for caches that have a fixed lifetime or are used for infrequently accessed data. Manual eviction, on the other hand, may be more appropriate for caches that store sensitive or frequently changing data.

The first point pertains to a configuration matter, and we'll examine the process in Section 4.2.12 on a local cache.

A method annotated with `@CacheEvict`[10] can delete entries from the cache:

```
@CacheEvict( cacheNames = "date4u.jsonhotprofiles" )
public void removeHotAsJson( List<Long> ids )
```

Again, the cache name is given in the annotation, and the parameter list is as known.

If we call the method from outside, the cache will remove the element. With the assignment `allEntries=true`, all elements for the named cache can be deleted.

```
@CacheEvict( cacheNames = "date4u.jsonhotprofiles",
             allEntries = true )
public void removeAllHotAsJson()
```

The parameter list is empty because a parameter isn't necessary. `condition` is also allowed with this annotation.

---

[10] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/cache/annotation/CacheEvict.html*

Methods annotated with `@CacheEvict` are usually responsible for deleting data from an underlying data store. As a result, these methods should also remove the corresponding data from the cache.

4.2.9 Specify Your Own Key Generators

In a cache, a key is always associated with a value. For methods annotated with `@Cacheable`, `@CachePut`, or `@CacheEvict`, the cache proxy must generate a key from the provided arguments. If a method has a single parameter with a “friendly” key type such as `long` or `String`, the key formation is straightforward. However, key formation can become more complicated when dealing with methods with multiple parameters or custom key types.

The problem starts when a method declares no parameter or more than one parameter. In addition, there may be parameter types that are not good keys.

To generate a key from passed arguments of the method, Spring uses a `KeyGenerator`[11] (see Figure 4.2). The interface condenses the passed arguments of a method into an `Object`, which is the key for the cache.

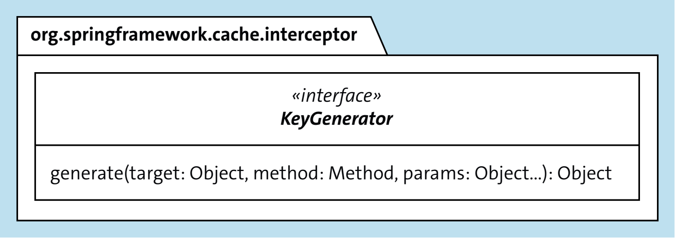

org.springframework.cache.interceptor
«interface»
KeyGenerator
generate(target: Object, method: Method, params: Object...): Object

Figure 4.2 The “KeyGenerator” Functional Interface

SimpleKeyGenerator

By default, the Spring Framework uses the `KeyGenerator` implementation `SimpleKeyGenerator` for mapping arguments to a cache key.[12] The code[13] is simple:

```
public class SimpleKeyGenerator implements KeyGenerator {

    @Override
    public Object generate(Object target, Method method, Object... params) {
        return generateKey(params);
    }
```

---

[11] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/cache/interceptor/KeyGenerator.html*

[12] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/cache/interceptor/SimpleKeyGenerator.html*

[13] *https://github.com/spring-projects/spring-framework/blob/main/spring-context/src/main/java/org/springframework/cache/interceptor/SimpleKeyGenerator.java*

```
    public static Object generateKey(Object... params) {
        if (params.length == 0) {
            return SimpleKey.EMPTY;
        }
        if (params.length == 1) {
            Object param = params[0];
            if (param != null && !param.getClass().isArray()) {
                return param;
            }
        }
        return new SimpleKey(params);
    }
}
```

The overridden method `generate(…)` from the interface calls its own public static method `generateKey(…)`, which does the actual mapping. `target` and the `Method` object aren't used by the `SimpleKeyGenerator`—the implementation only builds the cache key from the arguments of a method.

The method `generateKey(…)` considers four cases for the handover to the method intercepted by the cache proxy:

- The method has no parameters. This is rare, but valid, and, in this case, the return is `SimpleKey.EMPTY`. In this way, expensive factory methods can be declared whose results can be temporarily cached.
- The method has exactly one parameter, which is not `null` and also not an array. Then the passed argument really forms the key directly. This was the case with our list of IDs, and it's performant and memory-saving because no additional container objects are needed.
- The method has exactly one parameter, and it's `null`, or an array was passed as a data structure. Then a `SimpleKey` object is built with `null` or the array.
- If multiple parameters exist, a `SimpleKey` is generated for the arguments.

The SimpleKey Object

Often, the cache will reference a `SimpleKey`[14] object as the key. The code[15] looks like this (`readObject(…)` for serialization has been omitted):

```
package org.springframework.cache.interceptor;

import …
```

---

[14] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/cache/interceptor/SimpleKey.html*

[15] *https://github.com/spring-projects/spring-framework/blob/main/spring-context/src/main/java/org/springframework/cache/interceptor/SimpleKey.java*

```
public class SimpleKey implements Serializable {

    public static final SimpleKey EMPTY = new SimpleKey();


    private final Object[] params;

    private transient int hashCode;


    public SimpleKey(Object... elements) {
        Assert.notNull(elements, "Elements must not be null");
        this.params = elements.clone();
        this.hashCode = Arrays.deepHashCode(this.params);
    }


    @Override
    public boolean equals(@Nullable Object other) {
        return (this == other ||
                (other instanceof SimpleKey &&
                        Arrays.deepEquals(this.params,
                          ((SimpleKey) other).params)));
    }

    @Override
    public final int hashCode() {
        return this.hashCode;
    }

    @Override
    public String toString() {
        return getClass().getSimpleName() + " ["
        + StringUtils.arrayToCommaDelimitedString(this.params) + "]";
    }
    …
}
```

Listing 4.4 SimpleKey.java

The constructor accepts the method arguments passed to the proxy, copies the values, and calculates the hash value in advance so that the later query is cheap. The implementations of the `equals(…)` and `hashCode()` methods are important because the key must be found in the associative store. A `toString()` implementation is useful but unnecessary for the cache.

Set Your Own Key Generator via Key or keyGenerator

The `SimpleKeyGenerator` is a good standard, but it has disadvantages if, for example, `equals(…)/hashCode()` aren't overridden, aren't well implemented, or the object isn't desired as a key in the cache. Therefore, you can specify a key generator, and Spring provides two ways to do this.

The annotation types `@Cacheable`, `@CachePut`, and `@CacheEvict` provide two different ways to set a key generator via an annotation attribute:

```
public @interface Cacheable / CachePut / CacheEvict {
   …
  // Option 1:
  String key() default "";           (1)

  // Option 2:
  String keyGenerator() default "";  (2)
   …
}
```

Option 1 is a SpEL expression to extract a key; for option 2, we set the bean name of a Spring-managed bean, which implements the interface type `KeyGenerator`. We'll look at the details of option 2 in the next section.

Option 1 uses a SpEL expression to retrieve the key from the provided object. However, suppose a complicated object such as a book is involved. In that case, using the entire book as the cache key may be unnecessary and even unfeasible if there are no suitable `equals(…)/hashCode()` methods implemented. A more reasonable approach is to resort to the ISBN, which can serve as an excellent key for the object if it's a string.

Set the Key Generator via Key

Consider the type `Profil`, which represents a dating profile. While `Profile` objects should not be used as keys, a profile ID of type `long` serves as an excellent cache key.

```
@Cacheable( cacheNames = "…",
            key        = "#prof.id" )
Object method( Profile prof )
```

If a SpEL expression is specified for `key`, Spring will no longer use the `SimpleKeyGenerator` for key generation, but will fall back on the SpEL expression for key generation.

SpEL can make use of context objects, as we've already seen with `condition` and `unless`:

- `#root.args[0]`, `#p0`, `#a0`, or simply `#parameter-name`
- `#root.method`, `#root.methodName`, `#root.target`, `#root.targetClass`, and so on

Thus, `"#prof.id"` accesses the parameter `prof` and then the `id`. In the cache, the keys are of type `Long` (wrapper types) and not of type `Profile`.

Task: Set the Key Generator via Key

You can practice caching with the next task. As a reminder, `PhotoService` had a method `download(String)`:

```
public class PhotoService {
  …
  public Optional<byte[]> download( String name ) {
    try { return Optional.of( fs.load( name + ".jpg" ) ); }
    catch ( UncheckedIOException e ) { return Optional.empty(); }
  }
  …
}
```

Listing 4.5 PhotoService.java

In `PhotoService`, another overloaded method `download(Photo)` is to be added, so that callers can decide whether they want to get the image via the file name or via the `photo` (`Photo` contains the file name). `Photo` is a complex data type, and a key generator is needed:

```
public Optional<byte[]> download( Photo photo ) { implementMe }
```

The `Photo` parameter type is new and should look like this:

```
public class Photo {
  public Long          id;
  public Long          profile;
  public String        name;
  public boolean       isProfilePhoto;
  public LocalDateTime created;
}
```

Listing 4.6 Photo.java

Later, we'll extend the class so that photos can also be stored in the database.

A SpEL key generator is to be implemented that extracts `name` from `Photo` as the cache key.

The proposed solution follows:

```
@Cacheable( cacheNames = "date4u.filesystem.file",
            key        = "#photo.name" )
public Optional<byte[]> download( Photo photo ) {
  return download( photo.getName() );
}
```

Listing 4.7 PhotoService.java Extension

The parameter variable is called `photo`, and the SpEL expression `#photo.name` thus references the name, which is a string. Internally, the cache thus forms associations between the file name and the `byte` array.

Set the Key Generator via KeyGenerator

Returning to option 2, the custom `KeyGenerator` implementation, it's worth noting that although SpEL expressions are convenient and compact, they are less efficient and not as well-typed as direct Java code.

The `KeyGenerator` interface prescribes this method:

```
Object generate(Object target, Method method, Object... params)
```

The following example returns a `KeyGenerator` implementation via a `@Bean` method inside a `@Configuration` class:

```
@Bean
public KeyGenerator photoNameKeyGenerator() {
  return ( Object __, Method ___, Object... params ) -> {
    if ( params.length == 1 && params[ 0 ] instanceof Photo photo )
      return photo.name;
    throw new UnsupportedOperationException(
        "Can't apply this KeyGenerator here" );
  };
}
```

Because `KeyGenerator` is a functional interface, it can be implemented concisely using a lambda expression. From the parameter list, `Object` target and `Method` method aren't needed, only `params`, for the method arguments passed to the cache proxy.

If the `KeyGenerator` is used incorrectly, a consistency check follows: `params.length` must be `1`, and the type must be `Photo`. If this is true, the name is extracted and returned; otherwise, an exception follows.

The `KeyGenerator` implementation isn't set via a `Class` object, but via the bean name; the method is called `photoNameKeyGenerator`, and this is also the name of the component. It's set in the annotation attribute `keyGenerator`:

```
@Cacheable( cacheNames   = "date4u.filesystem.file",
            keyGenerator = "photoNameKeyGenerator" )
public Optional<byte[]> download( Photo photo )
```

Because `key` and `keyGenerator` are both strings, there is a danger of confusion.

4.2.10 @CacheConfig

If specifications like the cache name apply to all methods of a class, code duplication at each annotated caching method isn't optimal. Another solution is provided by the `@CacheConfig`[16] annotation, which is valid on one type and determines the name for `cacheNames`.

Here's an example:

```
@Component
@CacheConfig( cacheNames = "date4u.jsonhotprofiles" )
class HotProfileToJsonConverter {
  @Cacheable
  public String hotAsJson( List<Long> ids ) { … }
}
```

The cache name can be omitted for the individual methods. However, methods could also reassign the name.

In total, the `@org.springframework.cache.annotation.CacheConfig` annotation type declares four annotation attributes:

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface CacheConfig {
    String[] cacheNames() default {};
    String keyGenerator() default "";
    String cacheManager() default "";
    String cacheResolver() default "";
}
```

4.2.11 Cache Implementations

Up to this point, we've enabled caching declaratively using annotations, but we haven't yet discussed the actual implementation of the cache. By default, Spring uses the `java.util.concurrent.ConcurrentHashMap` data structure, which is a specialized associative store based on hash tables that allows concurrent modifications. However, this data structure isn't a true cache because it lacks an important property: a good cache should forget elements that haven't been used for a long time. In contrast, a `ConcurrentHashMap` would never forget values.

---

[16] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/cache/annotation/CacheConfig.html*

Distributed versus Local Caches

There are numerous cache implementations on the market, which can be categorized as follows:

- **Distributed cache**
  Data is stored on a cache server and not locally. All applications share the data on the cache server. Typical products in use are *Redis*, *Ehcache*, *Hazelcast*, *Infinispan*, *Couchbase*.
- **Local cache**
  Each application has its own cache in main memory. Typical Java libraries are *Caffeine* and *cache2k*.

Spring recognizes what is present based on the entry in the classpath via auto-configuration, and the proxy forwards the cache operations to the implementations. Often the lifetime, size, and so on can be configured externally via configuration properties, so that your own code doesn't notice any change of the cache implementation.

4.2.12 Caching with Caffeine

To illustrate a cache implementation, consider the open-source library Caffeine just mentioned (*https://github.com/ben-manes/caffeine*), which evolved from the *Google Guava cache*.

Two dependencies are necessary in the project object model (POM):

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-cache</artifactId>
</dependency>
<dependency>
  <groupId>com.github.ben-manes.caffeine</groupId>
  <artifactId>caffeine</artifactId>
  <version>3.1.8</version>
</dependency>
```

Listing 4.8 pom.xml Extension

`spring-boot-starter-cache` combines two dependencies, `spring-boot-starter` and `spring-contextsupport`, and has no code itself.

Next, the cache must be configured; this is possible, for example, via *application.properties*:

```
spring.cache.caffeine.spec=maximumSize=1,expireAfterWrite=10s
```

The key `spring.cache.caffeine.spec` contains a string with multiple segments separated by commas. One of these segments is the `maximumSize`, which can be set to `1` for testing purposes, meaning that only one element can be stored in the cache at a time. Additionally, the `expireAfterWrite` segment is set to `10` seconds, which causes any cached entry to automatically expire and be removed from the cache after 10 seconds have passed. It's worth noting that setting the `maximumSize` to `0` would disable caching altogether.

Note

To explore the configurations in detail, go to *www.javadoc.io/doc/com.github.benmanes.caffeine/caffeine/latest/com.github.benmanes.caffeine/com/github/benmanes/caffeine/cache/CaffeineSpec.html*.

In short, the following can be set:

- `initialCapacity=[integer]`
- `maximumSize=[long]`
- `maximumWeight=[long]`
- `expireAfterAccess=[duration]`
- `expireAfterWrite=[duration]`
- `refreshAfterWrite=[duration]`
- `weakKeys`
- `weakValues`
- `softValues`
- `recordStats`

If we make no configuration, a default configuration applies.

4.2.13 Disable the Caching in the Test

In the test case, caching is usually switched off. There are various solution approaches that can also be transferred to other technologies:

- **Option 1**
  Mostly there is a switch that can turn something off or on. `spring.shell.interactive.enabled=false` will turn off the shell, and `spring.main.banner-mode=off` can turn off the banner. For caching, this configuration property is `spring.cache.type`:

  ```
  spring.cache.type=NONE
  ```

  The assignment with `NONE` switches off the caching.

- **Option 2**

  Spring Boot enables a `CacheManager`[17] via auto-configuration. If there is no cache manager, Spring builds an instance. If we now build a cache manager ourselves, auto-configuration has nothing to do, and there is no cache:

  ```
  @Bean
  CacheManager cacheManager() {
    return new NoOpCacheManager();
  }
  ```

  The factory method is as usual in a `@Configuration` class and returns the `NoOpCacheManager`. As the name says—`NoOp`(eration)—nothing happens.

- **Option 3**

  Caching needs a proxy generator, and this is enabled by `@EnableCaching`. If the cache is to be disabled, this can be controlled by enabling `@EnableCaching`. For example, a profile could activate `@Configuration`. For example, if the application isn't in development mode, that is, it's in productive mode, this configuration should first be activated with `@EnableCaching`:

  ```
  @Profile( "!dev" )
  @EnableCaching
  @Configuration
  public class CachingConfig { }
  ```

  If one is in development mode, the profile isn't activated, so there is also no configuration. Without configuration, there is also no `@EnableCaching`—so no cache.

4.3 Asynchronous Calls

Concurrency is an essential aspect of modern programming that enables high-performance and efficient applications. The Spring Framework provides a powerful mechanism to use this capability through its proxy functionality, which allows for the execution of operations in a separate background thread. This feature enables the main program to continue its work while the background thread processes the task, leading to faster and more responsive applications.

To implement Spring proxies in your projects, you need to define the beans and configure the proxy settings in your application context. Once configured, Spring will automatically generate the proxy instances that you can use in your code. It's also possible to customize the proxy behavior by specifying additional properties, such as concurrency level, thread pool size, and synchronization mode.

In this section, we'll explore the benefits of using Spring proxies for concurrent programming and how to implement them in your projects.

---

[17] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/cache/CacheManager.html*

4.3.1 @EnableAsync and @Async Methods

As usual, the async proxy must be enabled; for this, a `@Configuration` is annotated with `@EnableAsync`.[18] In the next step, methods are annotated with `@Async`.[19] Here's an example:

```
@Async
public void heavyWork()
```

If we get a reference to the proxy wrapping the `heavyWork(…)` method, a call will by default cause the proxy to use a background thread and the `heavyWork(…)` method to work concurrently.

Asynchronous methods can have two different returns:

- In the simple case, you don't deliver anything back. This is also called the *fire-and-forget strategy*. That is, the return type is `void`.
- A method executed in the background can return a result, but a special return type must be used for this—a kind of container—because the receiver doesn't know when it will get the return. The answer may come 10 milliseconds later or a day later. Therefore, the `@Async` methods usually have the `Future` return type or the `CompleteableFuture` subtype. Via the `Future` objects, the client receives the result later when it's available.

Note: Language Comparison

While JavaScript has an `async` keyword for handling concurrency, Java lacks native language support for this feature. Although the Java SE library provides some concurrency tools, Spring's proxy functionality allows for the use of `@Async` methods that are almost as concise and user-friendly as their JavaScript counterparts. Additionally, implementing proxies through a library offers more flexibility for configuration compared to building the functionality directly into the language.

4.3.2 Example with @Async

Consider an example where both scenarios are combined: an `@Async` method with and without a return value.

```
@Component
class SleepAndDream {

  private final Logger log = LoggerFactory.getLogger( getClass() );
```

---

[18] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/scheduling/annotation/EnableAsync.html*

[19] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/scheduling/annotation/Async.html*

```
  @Async
  public void sleep1Seconds() throws Exception {
    log.info( "Going to sleep: SNNNNOOORRRREEEEE" );
    TimeUnit.SECONDS.sleep( 1 );
    log.info( "Woke up" );
  }

  @Async
  public CompletableFuture<String> sleep1SecondWithDream()
      throws Exception {
    log.info( "Starting to dream" );
    TimeUnit.SECONDS.sleep( 1 );
    log.info( "Finished the dream" );
    return CompletableFuture.completedFuture( "about unicorns" );
  }
}
```

The two methods are annotated with `@Async` and are `public`; this is important for the proxy generator. Something else is important too: if the methods call each other, there is no proxy in between.

Following are some details about each methods:

- **First method: `sleep1Seconds(…)`**
  This method outputs a logger message, sleeps for 1 second, so to speak, pauses the executing thread for 1 second, and writes a log message again. The `sleep(…)` method from the `TimeUnit` is a viable alternative to the `Thread.sleep(…)` method. Both `sleep(…)` methods can throw an `InterruptedException`, which the code passes up to the framework.
- **Second method: `sleep1SecondWithDream()`**
  This method has a return and uses the `Future` types. The code is similar to the first method, only at the end, the method builds, fills, and returns a container of type `CompletableFuture`. A `Future` object is somewhat similar to `Optional`, except it can take time to fill with data.

The `SleepAndDream` component can be injected; strictly speaking, it's the async proxy we get:

```
@Autowired SleepAndDream dreamer;
```

The next example calls the methods and therefore sets log messages. The comment at the beginning of the line indicates the threads that are active before the line is executed. `M` stands for the main thread, and `1` and `2` exemplarily for the threads used by the framework.

```
/*M  */ log.info( "Before sleep1Seconds()" );
/*M  */ dreamer.sleep1Seconds();
/*M1 */ log.info( "After sleep1Seconds()" );

/*M1 */ log.info( "Before sleep1SecondWithDream()" );
/*M1 */ Future<String> future = dreamer.sleep1SecondWithDream();
/*M12*/ log.info( "After sleep1SecondWithDream()" );

/*M12*/ TimeUnit.SECONDS.sleep( 2 );

/*M  */ log.info( "Dream: " + future.get() );
```

In simple Spring boot applications, the main thread executes the code; we see an `M` at the beginning. After the call to `sleep1Seconds(…)`, a short time later, there are two threads of action: one from the main thread and one from the (presumably newly built) thread that processes `sleep1Seconds()`. The main thread continues to run, unimpressed, the whole time. The thread reserved for `sleep1Seconds(…)` hangs in the method for 1 second and logs.

If the client on the proxy calls the second method, `sleep1SecondWithDream()`, then after a few milliseconds, another thread is used/created. This one is also condemned to sleep for 1 second. That is, two threads are in the waiting state, and the main thread also runs into the `SECONDS.sleep(2)`. Because the other two threads each take 1 second to complete, after 2 seconds, the result is definitely in the `Future` object and can be fetched. If the `SECONDS.sleep(2)` didn't exist, the output would be the same because the `get()` method is blocking and waiting. That is, if the result is in the container, it can be consumed directly, but `get()` blocks until the result is available. Take a second to think about what the program output might look like.

Note

In Spring, when a task needs to be executed, it uses threads from a thread pool instead of creating a new thread every time. This allows for more efficient usage of system resources and can reduce the overhead associated with creating new threads. The thread pool contains a predetermined number of threads that are available for processing tasks. Because the threads are being recycled, it can be difficult to determine whether a new thread or a previously used thread is being used for a specific task. The decision of which thread to assign a task to depends heavily on the current state of the thread pool. If all threads are currently busy, the task will have to wait until a thread becomes available. Once a thread is assigned a task, it will execute the task until completion. After the task has been executed, the thread becomes free again and is put back into the thread pool. At this point, the thread goes into a sleep state, waiting for the next instruction block to be assigned to it. This way, threads are reused efficiently, and the overhead associated with creating new threads is minimized.

Possible Output

The main thread will briskly do all the log output and bump the other threads:

```
… [   main] com.tutego.boot.Sleepyhead    : Before sleep1Seconds()
… [   main] com.tutego.boot.Sleepyhead    : After sleep1Seconds()
… [   main] com.tutego.boot.Sleepyhead    : Before sleep1SecondWithDream()
… [   main] com.tutego.boot.Sleepyhead    : After sleep1SecondWithDream()
… [task-2] com.tutego.boot.SleepAndDream : Starting to dream
… [task-1] com.tutego.boot.SleepAndDream : Going to sleep: SNNNNOOORRRREEEEE
… [task-1] com.tutego.boot.SleepAndDream : Woke up
… [task-2] com.tutego.boot.SleepAndDream : Finished the dream
… [   main] com.tutego.boot.Sleepyhead    : Dream: about unicorns
```

Until the processing and logging in the threads starts, it takes some time. The output could also look different because threads are highly nondeterministic. In the square brackets, you can easily read the thread names: `main`, `task-1`, and `task-2`.

Even if the threads were requested in a fixed order, they need exclusive access to the logger, and then `sleep1Seconds()` or else `sleep1SecondWithDream()` could win. After 1 second, almost both are finished at the same time, so there is again a race to the logger, so the order of the outputs could be different in principle.

4.3.3 The CompletableFuture Return Type

An asynchronous method in Spring returns a `Future` object, and here the `CompletableFuture` implementation is ideal.[20] The `Future` type is boring and has few methods: to fetch the result, to cancel, or to query its status. To obtain the result, we must use the `get(…)` method, as there are no other alternatives available.

An issue with the `Future` data type is that it doesn't support cascading, which means that if the `Future` contains a result, it can't trigger another operation in sequence. Developers familiar with programming in JavaScript may be familiar with the Promise application programming interface (API), which supports cascading through the use of the `then(…)` method, allowing for subsequent operations to be triggered.

The Java library has something similar, namely the extensive data type `CompletableFuture` as an implementation of `Future`. This can also be used to build cascades of operations, trigger background operations, return the results to the series, and so on. `CompletableFuture` is a "better" `Future`, and that is why we use it in the sample program.

The receiver can cascade operations and/or wait for the result with `get()`/`join()`. Schematically, it looks like this:

```
XXX xxx = completableFuture.thenApply(function).join();
```

---

[20] Before Spring Framework 6.0, people used `AsyncResult<V>`, but the type is now *deprecated*.

The `thenApply(…)` method takes the value from the `CompletableFuture`, calls a function with the value, and produces a new element in the pipeline. The `join(…)` method waits until the operation is complete.

Transmit Exceptions

During the execution of a background operation, unforeseen issues may arise, resulting in the throwing of an exception. Because the background operation is executed in a separate thread from the caller thread, any exceptions that occur may go unnoticed, and the background thread will terminate abruptly without returning any results. To prevent this scenario, exceptions must also be reported to the caller thread, allowing for proper handling of the error and ensuring that the client doesn't wait indefinitely for a result.

This is why there is a second static method besides `CompletableFuture.completedFuture(…)`, which is `failedFuture(Throwable ex)`. This way, the exception can be stored in the `CompletableFuture`, and it will show up again later when calling `get(…)` because if an exception is stored, `get(…)` will also throw an exception.

Suppose the thread has reported an exception with `CompletableFuture.failedFuture(…)`. This is how a client can react to the exception:

```
try {
  log.info( "{}", future.get() );
}
catch ( ExecutionException e ) {
  log.error( "{}", e.getClass() );
  // class java.util.concurrent.ExecutionException

  log.error( "{}", e.getCause() );
  // class of the original exception
}
```

On the `Future` object, `get(…)` is called, but a value isn't returned. Instead, the `get(…)` method will throw an `ExecutionException`. This signals that there was an error in the background thread. The error is queried via `getCause(…)`: the `ExecutionException` piggybacks the actual exception, so to speak. Wrapped exceptions occur in many places, for example, when a checked exception is wrapped in a `RuntimeException`.

Tip

There are two drawbacks to using an asynchronous method that returns `void`. First, the client may not be aware that the operation is asynchronous, which could be intentional or unintentional. Second, and a more significant issue, is that there is no feedback on whether the operation succeeded or failed in the background, as `void` signifies a fire-

and-forget approach where any failures aren't detected. To mitigate these problems, it's advisable to use `Future<Void>` or `CompletableFuture<Void>` when implementing asynchronous methods. This way, the caller can receive feedback on whether there were any exceptions in the background thread and take appropriate action based on the result.

Task: Calculate Thumbnail Concurrently

After so much theory, now follows a practical task for the further development of the Date4u application. In the class `PhotoService`, the `upload(…)` method is to be rewritten in such a way that the preview image is built up in a background thread.

Tip

If you want to execute something concurrently, the concurrent operation is triggered as early as possible. The program is then busy with something else first, and if the result is needed, the concurrent program is waited for if necessary.

Suggested solution: If thumbnail images are built in the background, the first thing to do is to adjust the `Thumbnail` interface:

```
public interface Thumbnail {
  @Async
  Future<byte[]> thumbnail( byte[] imageBytes );
}
```

Listing 4.9 Thumbnail.java Extension

So, we added `@Async`. There is one peculiarity: if the interface carries `@Async`, this also applies to the implementation. This isn't true for all annotations; some don't carry over to the implementations.

The annotation `@Async` alone isn't enough. The asynchronous method entails further change:

- The return value of the method needs to be wrapped in a `Future` or `CompletableFuture`.
- The client must retrieve the result from the `Future` object.
- The `@Async` annotation must be enabled with a corresponding enabler in the configuration.

To begin, let's modify the implementation of the interface, specifically in `AwtBicubicThumbnail`.

```
@Service
public class AwtBicubicThumbnail implements Thumbnail {
  …
  @Override
  public Future<byte[]> thumbnail( byte[] imageBytes ) {
    try ( InputStream is = new ByteArrayInputStream( imageBytes );
          ByteArrayOutputStream baos = new ByteArrayOutputStream() ) {
      ImageIO.write( create( ImageIO.read( is ), 200, 200 ), "jpg", baos );
      log.info( "thumbnail" );
      return CompletableFuture.completedFuture( baos.toByteArray() );
    }
    catch ( IOException e ) {
      throw new UncheckedIOException( e );
    }
  }
}
```

Listing 4.10 AwtBicubicThumbnail.java Extension

The byte array is placed into a `Future` object at the end.

Next, let's look at the call site at `PhotoService`. We've said that the background operation should be triggered as early as possible.

```
public String upload( byte[] imageBytes ) {
  Future<byte[]> thumbnailBytes = thumbnail.thumbnail( imageBytes );

  String imageName = UUID.randomUUID().toString();

  // First: store original image
  fs.store( imageName + ".jpg", imageBytes );

  // Second: store thumbnail
  try {
    log.info( "upload" );
    fs.store( imageName + "-thumb.jpg", thumbnailBytes.get() );
  }
  catch ( InterruptedException | ExecutionException e ) {
    throw new IllegalStateException( e );
  }

  return imageName;
}
```

Listing 4.11 PhotoService.java Extension

The generation of the preview image is initiated first in a background thread, and the Universally Unique Identifier (UUID) for the image name can be determined later.

However, it's possible for the background thread to throw an exception, requiring the use of a `try-catch` block. Passing the exception up to the caller would be unfair as it would create a problem for them. In our case, an `InterruptedException` can't occur, but an `ExecutionException` is always thrown when an error occurs in the background thread.

However, the program isn't yet fully functional as we haven't accounted for what happens if the preview image can't be generated. One solution could be to delete the original image as well, within a transaction. In the proposed solution, the exception is passed to the framework without any special handling.

To complete the program, we still need to configure the enabler.

```
@SpringBootApplication
@EnableCaching
@EnableAsync
public class Date4uApplication {
  public static void main( String[] args ) {
    SpringApplication.run( Date4uApplication.class, args );
  }
}
```

Listing 4.12 Date4uApplication.java Extension

Upon program start, the following threads are involved: the Spring Shell calls the method of the shell component, which then calls the `upload(…)` method of the service. This is executed on the main thread. On the other hand, the `thumbnail(…)` implementation is executed by a separate thread, as indicated by the logger. It should be noted that scaling the image is a relatively expensive operation, whereas saving it's generally cheaper because the program usually spends some time in the `get()` method waiting for the thumbnail to be ready.

4.4 TaskExecutor *

The asynchronous operations in the background must be executed by one entity. This is the job of an executor. To abstract this executor, Spring has declared the data type `TaskExecutor`.[21] A `TaskExecutor` inherits from `Executor` in the Java SE library (see Figure 4.3).

From today's perspective, a `TaskExecutor` would not be necessary, but the Java standard library only introduced the `Executor` type in Java 5. The `TaskExecutor` from Spring is older.

---

[21] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/core/task/TaskExecutor.html*

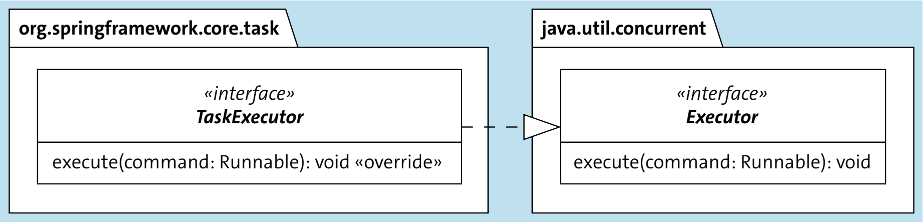

org.springframework.core.task
«interface»
TaskExecutor
execute(command: Runnable): void «override»
java.util.concurrent
«interface»
Executor
execute(command: Runnable): void

Figure 4.3 “TaskExecutor”: The "Runnable" Executor from Spring

Both interfaces declare method `execute(Runnable)`, so they can execute a code block of type `Runnable`. Which strategy is used to execute the `Runnable` is up to the implementation. Often a thread pool is used, but the `Runnable` could also be executed by a chosen thread. This is often used in the GUI environment, for example, where a block of code is to be executed in the GUI thread. It’s all a matter of the `TaskExecutor` implementation.

4.4.1 TaskExecutor Implementations

The Spring Framework provides a number of implementations of the `TaskExecutor` interface (see Figure 4.4).

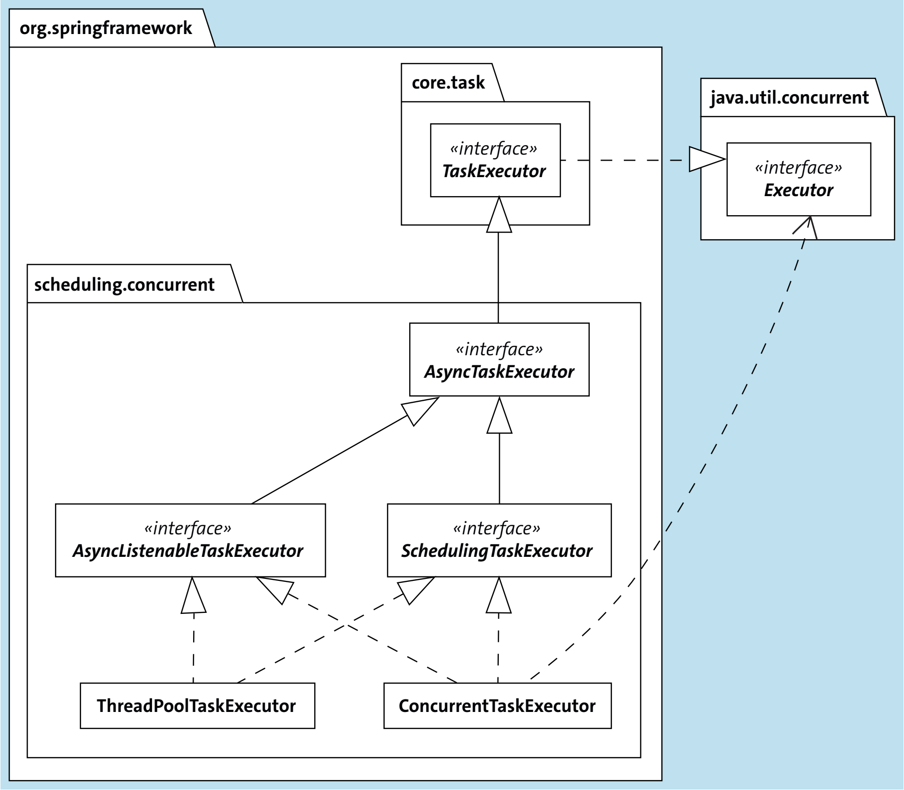

org.springframework
core.task
«interface»
TaskExecutor
java.util.concurrent
«interface»
Executor
scheduling.concurrent
«interface»
AsyncTaskExecutor
«interface»
AsyncListenableTaskExecutor
«interface»
SchedulingTaskExecutor
ThreadPoolTaskExecutor
ConcurrentTaskExecutor

Figure 4.4 “TaskExecutor” Implementations

Let's turn our attention to two implementations: the `ThreadPoolTaskExecutor` and the `ConcurrentTaskExecutor`. The `ThreadPoolTaskExecutor`[22] is a kind of predecessor of the current `java.util.concurrent.ThreadPoolExecutor`,[23] which works internally with a pool of threads. The `ThreadPoolTaskExecutor` is more powerful than the `ThreadPoolExecutor` of Java SE because the Spring thread pool can be reconfigured at runtime. Often, a thread pool is set up early at application startup and runs as long as the application. However, usage might deviate from the planned usage pattern and initial configuration, and then it's useful if, for example, the number of concurrent threads can be changed at runtime. The Java SE thread pool can't do this.

Another useful data type is the `ConcurrentTaskExecutor`, which brings an existing `Executor` implementation from Java SE into the Spring universe. The `ConcurrentTaskExecutor` is an application of the *adapter pattern*, which adapts two incompatible interfaces to each other, with the core functionality being the same. Spring 6.1 introduces support for a `TaskExecutor` featuring virtual threads through the `VirtualThreadTaskExecutor`.

Declare TaskExecutor Beans and Use Them for @Async

When using asynchronous operations with the `@Async` annotation, a `TaskExecutor` can be specified for execution. To accomplish this, a Spring-managed bean must be created and named accordingly:

```
@Bean( "threadPoolTaskExecutor" )
public TaskExecutor myThreadPoolTaskExecutor() {
  ThreadPoolTaskExecutor executor = new ThreadPoolTaskExecutor();
  Executor.…
  executor.initialize();
  return executor;
}

@Bean( "concurrentTaskExecutor" )
public TaskExecutor myConcurrentTaskExecutor () {
  return new ConcurrentTaskExecutor( Executors.newFixedThreadPool(3) );
}
```

The name is explicitly set, but, of course, the method name would be fine too.

The first `TaskExecutor` uses the Spring-specific type `ThreadPoolTaskExecutor`, and the second uses the Java SE `ThreadPoolExecutor`, which is adapted into a Spring data type `TaskExecutor` via `ConcurrentTaskExecutor`.

In the second step, the `@Async` annotation specifies the name of the `TaskExecutor` bean:

---

[22] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/scheduling/concurrent/ThreadPoolTaskExecutor.html*

[23] *https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/concurrent/ThreadPoolExecutor.html*

```
@Async( "threadPoolTaskExecutor" ) public void abc() { }
@Async( "concurrentTaskExecutor" ) public void xyz() { }
```

This allows the asynchronous calls to be processed with differently configured executors.

4.4.2 Set Executor and Handle Exceptions

When Spring uses a `TaskExecutor` for execution, an exception may occur in the `Runnable`. The question then becomes: What happens to the exception? We've looked at our dream method in Section 4.3.2, which showed that with `@Async` methods, even checked exceptions can be forwarded to the framework.

What should happen with the exceptions that arrive at the framework can be configured via an `AsyncConfigurer`.[24] The interface looks like this.

```
package org.springframework.scheduling.annotation;

import …

public interface AsyncConfigurer {

  @Nullable
  default Executor getAsyncExecutor() { return null; }

  @Nullable
  default AsyncUncaughtExceptionHandler getAsyncUncaughtExceptionHandler() {
    return null;
  }
}
```

Listing 4.13 AsyncConfigurer.java

The `AsyncConfigurer` provides two objects: an `Executor` that executes the code, and an `AsyncUnchaughtExceptionHandler` for the uncaught exceptions. The `AsyncUncaughtExceptionHandler` data type also comes from the Spring environment.

```
package org.springframework.aop.interceptor;

import java.lang.reflect.Method;

@FunctionalInterface
public interface AsyncUncaughtExceptionHandler {
```

---

[24] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/scheduling/annotation/AsyncConfigurer.html*

```
 void handleUncaughtException(Throwable ex, Method method,
                              Object... params);
}
```

Listing 4.14 AsyncUncaughtExceptionHandler.java

In Java SE, there is something similar called `UncaughtExceptionHandler`.[25] The difference between the two is that in the Java SE environment, an `UncaughtExceptionHandler` is used only for unchecked exceptions because everything else has already been caught with `Runnable` according to the API contract (the `run()` method of `Runnable` has no `throws`). Conversely, in the Spring environment, the `AsyncUncaughtExceptionHandler` can handle all exceptions. The default implementation of Spring is the `SimpleAsyncUncaughtExceptionHandler`.[26]

```
public class SimpleAsyncUncaughtExceptionHandler implements AsyncUncaughtExcepti
onHandler {

    private static final Log logger =
       LogFactory.getLog(SimpleAsyncUncaughtExceptionHandler.class);


    @Override
    public void handleUncaughtException(Throwable ex, Method method,
                                        Object... params) {
        if (logger.isErrorEnabled()) {
            logger.error(
               "Unexpected exception occurred invoking async method: "
              + method, ex);
        }
    }
}
```

Listing 4.15 SimpleAsyncUncaughtExceptionHandler.java

From the implementation, you can see that the exception is logged and nothing else happens.

A Custom AsyncConfigurer Implementation

If a custom executor and an `AsyncUncaughtExceptionHandler` need to be specified, an `AsyncConfigurer` implementation can be defined as a `@Configuration`, which might look like this:

---

[25] *https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/Thread.UncaughtExceptionHandler.html*

[26] *https://github.com/spring-projects/spring-framework/blob/main/spring-aop/src/main/java/org/springframework/aop/interceptor/SimpleAsyncUncaughtExceptionHandler.java*

```
@Configuration
class AsyncConfig implements AsyncConfigurer {
  private final Logger log = LoggerFactory.getLogger( getClass() );

  @Override
  public Executor getAsyncExecutor() {
    ThreadPoolTaskExecutor executor = new ThreadPoolTaskExecutor();
    // …
    return executor;
  }

  @Override
  public AsyncUncaughtExceptionHandler getAsyncUncaughtExceptionHandler() {
    return ( throwable, method, params ) -> {
      log.info( "Exception: {}", throwable );
      log.info( "Method: {}", method );
      IntStream.range( 0, params.length ).forEach(
        index -> log.info( "Parameter {}: {}", index, params[ index ] )
      );
    };
  }
}
```

The `AsyncConfigurer` implementation overrides both `default` methods. For the `Executor`, the `getAsyncExecutor()` method builds a `ThreadPoolTaskExecutor`. `getAsyncUncaughtExceptionHandler()` returns an implementation of the `AsyncUncaughtExceptionHandler` functional interface via a lambda expression. The data received from the parameter list (`Throwable ex, Method method, Object... params`) are all logged.

4.5 Spring and Bean Validation

Validating the state of objects is an essential aspect of software development, as it helps ensure that data is consistent and accurate throughout the application. In this section, we'll explore how to use Spring and Jakarta Bean Validation to implement robust validation rules and improve the reliability of your application.

4.5.1 Parameter Checks

Typical methods always have the same structure: first, they check the validity of parameters, and then only continue if everything is okay. Normally, incorrect parameters are reported by an `IllegalArgumentException`, and sometimes also by a `NullPointerException`. In the following, we see two examples from Java SE:

- **Example 1:** `ArrayList/get(…)`

  ```
  public E get(int index) {
    Objects.checkIndex(index, size);
    return elementData(index);
   }
  ```

  The index for the access must not be negative and not too large. `elementData(…)` is only called when the parameter assignment is okay, and the result is returned.

- **Example 2:** `LocalDate.of(…)`

  ```
  public static LocalDate of(int year, Month month, int dayOfMonth) {
    YEAR.checkValidValue(year);
    Objects.requireNonNull(month, "month");
    DAY_OF_MONTH.checkValidValue(dayOfMonth);
    return create(year, month.getValue(), dayOfMonth);
   }
  ```

  Parameter checking can also be found in constructors and factory methods, for example, in the `of(…)` method of `LocalDate`.

Often, methods are divided into two sections: validation of parameter assignments at the top, followed by the actual logic that can assume that all parameter variables are correct. However, if all validations are placed at the beginning of the method, it may be worth considering whether certain checks can be deferred to the framework. This is because anything that occurs at the beginning or end of a method can be handled by a proxy.

4.5.2 Jakarta Bean Validation (JSR 303)

Jakarta Bean Validation is a standard with two parts at its core: annotations used to describe valid states, and a validator used to check the rules. Jakarta Bean Validation provides a number of basic annotation types, for example:

- `@Size`
- `@NotNull`, `@Null`
- `@Min`, `@Max`
- `@Pattern`
- `@Past`, `@Future`

New annotation types can be declared, which in turn can validate their own rules. The annotations are valid on instance variables, setters/getters, or method parameters.

4.5.3 Dependency on Spring Boot Starter Validation

Although the Spring Framework relies on validation in many places, the core starter doesn't include an implementation. If we want to use validation, an additional dependency is required.

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-validation</artifactId>
</dependency>
```

Listing 4.16 pom.xml Extension

The dependency contains the core starter and, as a reference implementation of the Jakarta Bean Validation, the *Hibernate Validator* (*https://github.com/hibernate/hibernate-validator*). Spring Boot 3 uses the latest Hibernate Validator 8 (Jakarta Bean Validation 3), whereas Spring Boot 2.7 uses only Hibernate Validator 6 (Jakarta Bean Validation 2).

The Hibernate Validator serves as the default implementation in Jakarta Enterprise Edition, which comes with its own set of benefits. First, it provides additional validation capabilities through proprietary annotation types, which allows developers to create more robust validation rules. Second, the use of the Hibernate Validator isn't limited to just Spring applications, as it's compatible with other enterprise frameworks as well. Therefore, developers can use the same validation framework across multiple projects and easily port Spring applications to other frameworks without any issues.

4.5.4 Photo with Jakarta Bean Validation Annotations

The Date4u application relies on various datasets such as login information, profile information, likes, and more to function as a dating application. Among these, the `Photo` class is used to store information about photos, which was introduced in Section 4.2.9. To ensure that invalid photo information can be easily detected, we can annotate the `Photo` datatype with validation annotations.

```
import jakarta.validation.constraints.*;
import java.time.LocalDateTime;

public class Photo {

  public Long id;
  @Min( 1 ) public long profile;
  @NotNull @Pattern( regexp = "[\\w_-]{1,200}" ) public String name;
  public boolean isProfilePhoto;
  @NotNull @Past public LocalDateTime created;

  public Photo() { }
  public Photo( … ) { … }
  @Override public String toString() { … }
}
```

Listing 4.17 Photo.java Extension

These annotations are applied to instance variables, and they are also valid for setters and getters. Let's review the configuration settings:

- **`profile`**
  The `long` variable stores the foreign key on a corresponding profile. The annotation states that the number must always be greater than or equal to `1`. As a primitive data type `long`, the variable can never be `null`.
- **`name`**
  The name of the photo is a `String` and must be given, that is, not `null`. The regex expression dictates that any word character, underscore character, or minus character is allowed, and the length must be between 1 and 200. The word characters include all digits and all uppercase/lowercase letters—but only the letters of the English alphabet from A to Z without special characters. Certainly, file names are likely to contain additional characters in practice. We should change this in a larger example.
- **`created`**
  For `LocalDateTime`, the value should not be `null` and must be in the past because photos can't be created in the future.

If the photo is a profile photo, the `boolean` variable `isProfilePhoto` is set. Here you can't validate anything because an annotation is missing.

The `id` instance variable of the `Photo` class is left without any validation annotation, as its assignment is dependent on the state of the object. A newly created `Photo` object doesn't have an assigned ID, which happens only after it's saved to the database. An ID assigned to a saved photo must be a number greater than or equal to 1. The Jakarta Bean Validation framework provides groups to handle cases where values must be validated differently in different scenarios.

Note

The annotation `@NotNull` is a necessary addition to `@Pattern` and `@Past`, because if the variable is `null`, the other checks aren't made.

We added a parameterized and parameterless constructor to `Photo` to make it easier to build the `Photo` object, as well as a `toString()` method. But this has nothing to do with validation.

Note

The annotation `jakarta.validation.constraints.`**`NotNull`** (for validation proxies) can easily be confused with `org.springframework.lang.`**`NonNull`** or `javax.annotation.`**`Nonnull`** (for static analysis tools).

The class in question doesn't possess any annotations, particularly the Spring annotation `@Component` or any similar ones. This object isn't meant to function as a Spring component, but instead serves as a simple container. Typically, these objects don't perform any significant actions and lack domain model intelligence or critical behavior. Rather, they exist solely for data exchange purposes, functioning as data classes.

IntelliJ Tip

IntelliJ Ultimate Edition has a special view for validators, which can be activated under **View • Tool Windows • Bean Validation**.

4.5.5 Inject and Test a Validator

Validation of objects can be done in two ways: declaratively or programmatically. We'll begin by exploring the programmatically driven approach.

Spring Boot automatically configures a Spring-managed bean of type `Validator`,[27] which we can inject:

```
@Autowired Validator validator;
```

The `Validator` interface focuses on `validate*(…)` methods, such as Figure 4.5 shows.

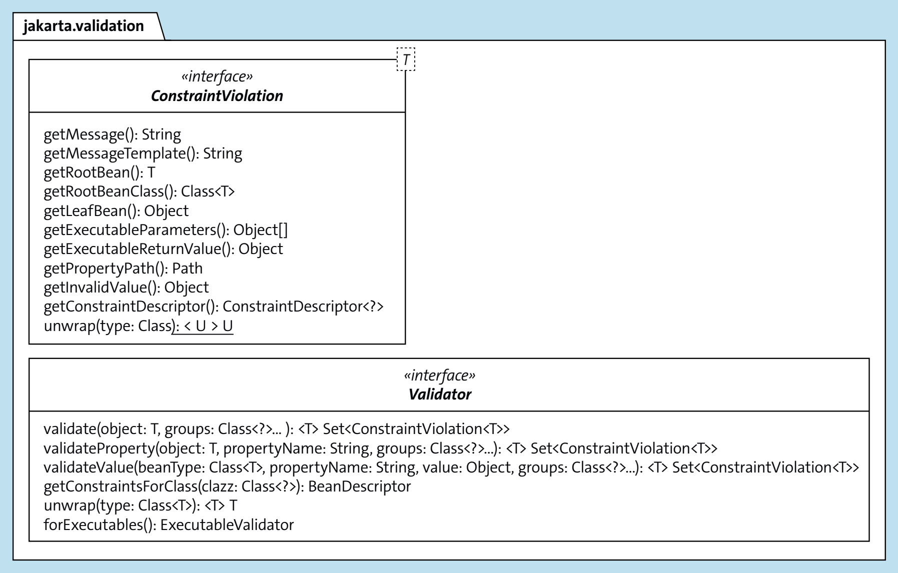

jakarta.validation
T
«interface»
ConstraintViolation
getMessage(): String
getMessageTemplate(): String
getRootBean(): T
getRootBeanClass(): Class<T>
getLeafBean(): Object
getExecutableParameters(): Object[]
getExecutableReturnValue(): Object
getPropertyPath(): Path
getInvalidValue(): Object
getConstraintDescriptor(): ConstraintDescriptor<?>
unwrap(type: Class): < U > U
«interface»
Validator
validate(object: T, groups: Class<?>...): <T> Set<ConstraintViolation<T>>
validateProperty(object: T, propertyName: String, groups: Class<?>...): <T> Set<ConstraintViolation<T>>
validateValue(beanType: Class<T>, propertyName: String, value: Object, groups: Class<?>...): <T> Set<ConstraintViolation<T>>
getConstraintsForClass(clazz: Class<?>): BeanDescriptor
unwrap(type: Class<T>): <T> T
forExecutables(): ExecutableValidator

Figure 4.5 "Validator" Triggering Validation and "ConstraintViolation" Representing Validation Error

---

[27] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/validation/Validator.html*

When invoking the `validate(…)` method, the object that needs to be validated is supplied as an argument. The resulting `Set` contains `ConstraintViolation`[28] objects, which indicate a problem if the set isn't empty. Conversely, an empty set implies that there are no validation errors.

Assume that a photo of the shell is to be entered into the database:

```
@Autowired Validator validator;

@ShellMethod( "Insert a new photo" )
String insertPhoto( Long id, long profile, String name,
                    boolean isProfilePhoto, String created ) {

  Photo photo = new Photo( id, profile, name, isProfilePhoto,
                           LocalDateTime.parse( created ) );

  Set<ConstraintViolation<Photo>> violationSet = validator.validate( photo );

  // Insert photo into database if violationSet.isEmpty()

  return violationSet.isEmpty() ? "Photo inserted"
                                : "Photo not inserted\n" + violationSet;
}
```

This reference to the `Validator` is used to perform the validation of the `Photo` object. The result of the validation is stored in a `Set` of `ConstraintViolation` objects. If the set is empty, there are no violations of the validation rules, and, theoretically, the photo can be inserted into the database. If there are validation errors, a different string is returned along with the details of the violations.

Let's call the shell method once with correct and with broken data:

```
shell:>insert-photo 12 1 bella false 2020-09-01T12:20
Photo inserted
shell:>insert-photo 12 0 üüüüä false 2222-09-01T12:20
Photo not inserted:
[ConstraintViolationImpl{interpolatedMessage=
'must be greater than or equal to 1', ↩
propertyPath=profile, rootBeanClass=
class com.tutego.date4u.interfaces.shell.Photo, ↩
messageTemplate='{jakarta.validation.constraints.Min.message}'}, ↩
ConstraintViolationImpl{interpolatedMessage='must match "[\w_-]{1,200}"', ↩
propertyPath=name, rootBeanClass=
class com.tutego.date4u.interfaces.shell.Photo, ↩
```

---

[28] *https://jakarta.ee/specifications/bean-validation/3.0/apidocs/jakarta/validation/constraintviolation*

```
messageTemplate='{jakarta.validation.constraints.Pattern.message}'}, ↩
ConstraintViolationImpl{interpolatedMessage='must be a past date', ↩
propertyPath=created, rootBeanClass=
class com.tutego.date4u.interfaces.shell.Photo, ↩
messageTemplate='{jakarta.validation.constraints.Past.message}'}]
```

In the second string, the `Validator` finds a number of problems:

- `profile` must not be 0 because, according to the rule, foreign keys must always be greater than or equal to 1.
- The `name` contains German umlauts. This is wrong because `\w` only covers uppercase/lowercase letters and digits, but not German umlauts.
- The timestamp `created` isn't in the past.

The `toString()` representation may of course be different depending on the implementation, but at first glance, more details become clear:

- **`interpolatedMessage`**
  This is an automatically generated error message in the respective local language.
- **`propertyPath`**
  This is the originator; in our case, these are `profile`, `name`, and `created`.
- **`rootBeanClass`**
  This is the `Photo` class with the invalid properties.
- **`messageTemplate`**
  This is the key for a translation table.

In summary, when a Jakarta Bean Validator implementation is present in the classpath, Spring auto-configuration constructs a `Validator` as a managed bean that can be injected by a client for use. The `Validator` includes a `validate(…)` method that returns a `Set<ConstraintViolation>` rather than an exception.

It all depends on how we utilize this information. Certain web frameworks, such as Jakarta Faces, are capable of directly binding the error messages to the GUI.

4.5.6 Spring and the Bean Validation Annotations

In addition to programmed validation, there is another way to use Jakarta Bean Validation through declarative validation using proxies that verify the validity of objects or parameters. This can be particularly useful in the `download(…)` method of the `PhotoService`, where only valid photos should be accepted.

```
@Service
@Validated
public class PhotoService {
  public Optional<byte[]> download( String imageName ) { … }

  public Optional<byte[]> download( @Valid Photo photo ) {
    return download( photo.name );
  }

  // …
}
```

Listing 4.18 PhotoService.java Extension

Two annotations are used:

- `@Validated`[29] from the `org.springframework.validation.annotation` package leads to a new proxy that works internally with a `Validator`.
- `@Valid`[30] from the `jakarta.validation` package before the parameter tells the proxy that a valid object must be passed; the type comes from Jakarta Bean Validation. If the annotation is missing, nothing is validated because it's fine in other scenarios to get empty photos and then initialize and fill them, for example.

If a program injects a `PhotoService`, the result is a validation-enabled proxy:

```
@Autowired PhotoService photoService;
```

The `download(Photo)` method undergoes a validation check through a proxy when called on this object. The proxy verifies if the photo obeys the rules specified for its validity. If the photo passes the validation, the proxy allows the caller to access the core logic. However, if the photo fails to meet the validation rules, a `ConstraintViolationException`[31] is thrown by the proxy.

This is how the call could look with a logging of the violations:

```
try {
  Photo photo = new Photo();
  photoService.download( photo );
}
catch ( ConstraintViolationException e ) {
  Set<ConstraintViolation<?>> violations = e.getConstraintViolations();
  log.info( violations.toString() );
}
```

---

[29] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/validation/annotation/Validated.html*

[30] *https://jakarta.ee/specifications/bean-validation/3.0/apidocs/jakarta/validation/valid*

[31] *https://jakarta.ee/specifications/bean-validation/3.0/apidocs/jakarta/validation/constraintviolationexception*

The newly created `Photo` object violates various rules, so the proxy validator will report the problem and throw a `ConstraintViolationException`—the unchecked exception type comes from Jakarta Bean Validation, not Spring. The `ConstraintViolationException` object returns the known set of small `ConstraintViolation` objects with `getConstraintViolations()`.

4.5.7 Bean Validation Annotations to Methods

Annotations from Jakarta Bean Validation can be directly used on both return values and parameters. Consider the following example:

```
@Service
@Validated
class NiceService {

  public @NotNull Object aValidMethod( @NotNull String param1,
                                       @Max(10) int    param2 ) {
    return "";
  }

}
```

A class such as `Photo` with annotated properties isn't strictly necessary, as the example shows: the proxy validator will check that the method doesn't return `null`, `arg1` wasn't `null` when called, and the values at `arg2` are at most 10.

Note

It's important to note that object validation only occurs through the proxies. If a program doesn't go through the proxies, for example, a self-programmed test case without Spring's testing infrastructure, the validation won't be executed. Therefore, if validation is critical to the application, it's recommended to program the validation check at the beginning of each method, regardless of whether it's also being checked by the proxy. This approach ensures that the validation is always performed, no matter which calling method is used.

4.5.8 Validation Everywhere Using a Configuration Example

Jakarta Bean Validation can be found in many places: in Jakarta EE as well as in the Spring environment. Popular examples are the *Jakarta Persistence API* for object-relational mapping, *Jakarta Faces* for dynamic web pages, or in *Jakarta RESTful Web Services*.

Spring Boot can also check configuration properties passed to an object via `@ConfigurationProperties` using Jakarta Annotations.

The Project Lombok and Jakarta Bean Validation technologies lead to the following compact example, where `GeometryProperties` references a `Box` and a `Circle` with data:

```
@Component @ConfigurationProperties( "app" ) @Data @Validated
class GeometryProperties {
  @NotNull @Valid private GeometryProperties.Box    box;
  @NotNull @Valid private GeometryProperties.Circle circle;

  @Data
  public static class Box {
    @Min( 50 ) @Max( 1000 ) private int width;
    @Min( 50 ) @Max( 600 )  private int height;
  }

  @Data
  public static class Circle {
    @Min( 10 ) @Max( 1000 ) private int radius;
  }
}
```

The `@Data` annotation[32] comes from *Project Lombok* (*https://projectlombok.org*). The open-source software is in essence a compiler hack that automatically puts setters, getters, parameterized constructors, loggers, and so on into the bytecode. This is always used if you want to save setter/getter and other "boilerplate code," but it's necessary in the bytecode—just for the `@ConfigurationProperties`.

If `@Data` is applied, you can imagine setter/getter in the following:

- **`GeometryProperties`**: `setBox(Box)`, `getBox()`, `setCircle(Circle)`, `getCircle()`
- **`Box`**: `setWidth(int)`, `getWidth()`, `setHeight(int)`, `getHeight()`
- **`Circle`**: `setRadius(int)`, `getRadius()`

The `GeometryProperties` object must reference `Box` and `Circle` because validation doesn't allow `null` assignments with `@NotNull`. In the subobjects, the properties would have to be in certain value ranges.

When Spring Boot has read the `Environment` data and transferred it to the bean, validation follows. If there is an error, the container doesn't start and reports the problem.

4.5.9 Test Validation *

Suppose we want to test the `download(Photo)` method. The test should show that there is no exception for a correct photo and that a broken, not valid photo leads to an exception and setting special error conditions.

---

[32] *https://projectlombok.org/features/Data*

The class `PhotoServiceTest` will get a nested class for the validator-related tests, and two new methods—`photo_is_valid()` and `photo_has_invalid_created_date()`—will check that an error-free `Photo` doesn't lead to an exception, but a broken photo does:

```
@SpringBootTest
class PhotoServiceTest {

  …

  @Nested
  class Validator {

    @Test
    void photo_is_valid() {
      Photo photo = new Photo( 1, 1, "fillmorespic", false,
                               LocalDateTime.MIN );
      assertThatCode( () -> photoService.download( photo ) )
          .doesNotThrowAnyException();
    }

    @Test
    void photo_has_invalid_created_date() {
      …
    }
  }
}
```

The first method, `photo_is_valid()`, is simple: it builds a `photo` with valid states and then passes it to the method `download(Photo)`. Actually, the test method could end here, but `assertThatCode(…).doesNotThrowAnyException()` makes it more explicitly readable that no exception must occur.

Note

The `PhotoService` will use `FileSystem`, and the `FileSystem` must also return something. Therefore, it's important that Mockito has the appropriate objects built correctly. How to build mock objects was shown in Chapter 3, Section 3.8.5.

The second method, `photo_has_invalid_created_date()`, shows that the validator reports a certain error in case of an incorrect date—which isn't in the past but in the future:

```
LocalDateTime future = LocalDateTime.of( 2500, 1, 1, 0, 0, 0 );
Photo photo = new Photo( 1L, 1L, "fillmorespic", false, future );
assertThatThrownBy( () -> photoService.download( photo ) )
```

```
  .isInstanceOf( ConstraintViolationException.class )
  .extracting(
    throwable ->
      ((ConstraintViolationException) throwable).getConstraintViolations(),
    as( InstanceOfAssertFactories.collection( ConstraintViolation.class ) )
  )
  .hasSize( 1 )
  .first( InstanceOfAssertFactories.type( ConstraintViolation.class ) )
  .satisfies( vio -> {
    assertThat( vio.getRootBeanClass() ).isSameAs( PhotoService.class );
    assertThat( vio.getLeafBean() ).isExactlyInstanceOf( Photo.class );
    assertThat( vio.getPropertyPath()).hasToString("download.photo.created");
    assertThat( vio.getInvalidValue() ).isEqualTo( future );
    // assertThat( vio.getMessage() ).isEqualTo( … );
  } );
```

While the readability may not be exceptional, the code example effectively showcases the capabilities of AssertJ. It's worth noting that using AssertJ doesn't always guarantee optimal readability, but it does provide a powerful toolset for writing comprehensive and expressive assertions.

The photo in year 2500 will result in the following:

1. A `ConstraintViolationException` will occur.
2. The set returned by `getConstraintViolations()` will contain exactly one element.
3. The element will be of type `ConstraintViolation`.
4. Of these, `getRootBeanClass()` will return a `PhotoService`, `getLeafBean()` will return `Photo`, and so on.
5. Because the message is always translated, that is, localized, this is a disadvantage when testing, and therefore no testing is done.

In practice, you would omit certain checks. For example, it's clear that `getConstraintViolations()` only contains `ConstraintViolation` objects, so the validation implementation must take care of that. The rule is "never test framework code."

Refactoring the Photo Class

Because we'll later use the `Photo class` as an entity bean, we refactor it so that the instance variables become private and end up with setters/getters:

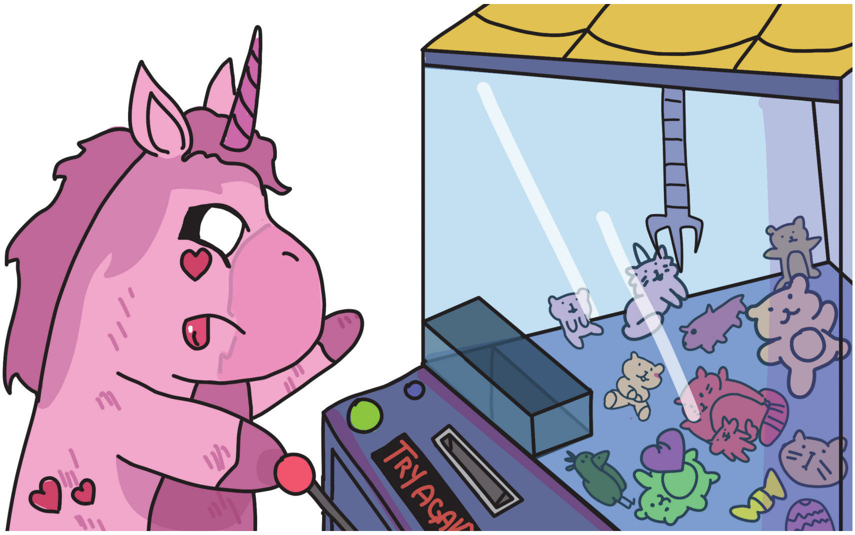
```
public class Photo {

  private Long id;

  @Min( 1 )
  private Profile profile;
```

```
 @NotNull @Pattern( regexp = "[\\w_-]{1,200}" )
 private String name;

 private boolean isProfilePhoto;

 @NotNull @Past
 private LocalDateTime created;

 protected Photo() { }

 public Photo( Long id, Profile profile, String name, boolean isProfilePhoto,
LocalDateTime created ) {
   this.id = id;
   this.profile = profile;
   this.name = name;
   this.isProfilePhoto = isProfilePhoto;
   this.created = created;
 }

 public Long getId() {
   return id;
 }

 public Profile getProfile() {
   return profile;
 }

 public String getName() {
   return name;
 }

 public void setName( String name ) {
   this.name = name;
 }

 public boolean isProfilePhoto() {
   return isProfilePhoto;
 }

 public void setProfilePhoto( boolean profilePhoto ) {
   isProfilePhoto = profilePhoto;
 }

 public LocalDateTime getCreated() {
    return created;
  }

  public void setCreated( LocalDateTime created ) {
    this.created = created;
  }

  @Override public String toString() {
    return "Photo[" + id + "]";
  }
}
```

This makes the `toString()` method a little simpler.

4.6 Spring Retry *

In today's interconnected world, most services are connected remotely, such as databases, messaging systems, and email servers. However, the connection to these servers can be unreliable, and the system may become unavailable. In such a scenario, simply timing out the program isn't a reliable solution as we expect some *resilience* from our system. To address this problem, one possible solution is for the client to wait a bit and then try again, as the network glitch may be resolved by then.



4.6.1 Spring Retry Project

In the Spring ecosystem, various approaches exist to tackle a problem. In this section, we'll explore a compelling proxy from the *Spring Retry project* (*https://github.com/*

spring-projects/spring-retry) that can rerun code blocks in case of exceptions. This feature proves beneficial for services that rely on network connections, as the network can experience brief outages due to topology changes or routing table adjustments.

To deploy the Spring Retry project, two dependencies are placed in the POM:

```
<dependency>
  <groupId>org.springframework.retry</groupId>
  <artifactId>spring-retry</artifactId>
</dependency>

<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-aspects</artifactId>
</dependency>
```

The first dependency references the Spring Retry project itself. The second dependency is from the area of AOP and introduces *Spring Aspects* so that proxies can be generated.

Job: Determine a Random Image via a Web Service

The *Random User Generator* is a website and web service (*http://randomuser.me*) that generates different sets of personal information. For instance, we aim to create a simple program that requests random images. To achieve this, we can customize the web service using parameters: *https://randomuser.me/api/?inc=picture&noinfo*. The API endpoint is available through both HTTP and HTTPS.

A call to the URL in the browser returns the following (formatted) JSON document:

```
{
  "results": [
    {
      "picture": {
        "large": "https://randomuser.me/api/portraits/men/0.jpg",
        "medium": "https://randomuser.me/api/portraits/med/men/0.jpg",
        "thumbnail": "https://randomuser.me/api/portraits/thumb/men/0.jpg"
      }
    }
  ]
}
```

Behind `results` is an array, and because only one picture was requested, there is only one `picture`. The picture is available in three different resolutions: `large`, `medium`, and `thumbnail`.

The image can be queried for one gender; then the query parameter `gender` can be supplemented with `female` or `male`:

- *https://randomuser.me/api/?inc=picture&noinfo&gender=female*
- *https://randomuser.me/api/?inc=picture&noinfo&gender=male*

A Java program is to call this endpoint and obtain the result. It should consider that there may be connection errors. In case of exceptions, Spring Retry should automatically invoke the web service again.

4.6.2 @Retryable

When attempting to retrieve a string containing JSON, the new `RandomPhoto` component throws an exception, simulating network errors.

```
@Component
class RandomPhoto {

  private final Logger log = LoggerFactory.getLogger( getClass() );
  private int attemptCounter = 1;

  @Retryable
  public String receive( String gender ) throws IOException {
   log.info( "Attempts: {}", attemptCounter++ );
   if ( attemptCounter <= 3 )
      throw new IOException( "Not yet ready to serve" );
   var url="https://randomuser.me/api/?inc=picture&noinfo&gender="+gender;
   try ( var inputStream = URI.create( url ).toURL().openStream() ) {
     return StreamUtils.copyToString( inputStream, StandardCharsets.UTF_8 );
   }
  }
}
```

Listing 4.19 RandomPhoto.java

The public `receive(…)` method is annotated with `@Retryable`,[33] and this tells the retry proxy to call the method again if an exception occurred. The method expects a gender as a parameter, and the return is the string with the JSON document.

The code takes three tries for a random image; thus, we simulate network/service failures. The client won't notice any of this.

The actual call of this web service is very simply implemented via the `URL` class. There are various classes for calling web services, such as the `HttpClient` of Java SE and the `WebClient` or the `RestTemplate` of Spring, but this is actually one with the shortest code.

---

[33] *https://docs.spring.io/spring-retry/docs/api/current/org/springframework/retry/annotation/Retryable.html*

Via the `URL` class, the program gets an `InputStream`, and the Spring utility class `StreamUtils` receives the data via `copyToString(…)` and puts it into a string.

@EnableRetry

For the `RandomPhoto` component, a retry proxy must be enabled that responds to `@Retryable`. For this purpose, `@EnableRetry`[34] is set to a `@Configuration` (like `@SpringBootApplication`). This is how it might look:

```
@Configuration
@EnableRetry
public class RetryConfig { }
```

This builds up corresponding retry proxy objects. If the client calls the `receive(…)` method via the retry proxy, the proxy intercepts the annotated methods and repeatedly goes to the core in case of exceptions.

Watching the Proxy Repeats

We can inject `RandomPhoto` into a client and get a retry proxy:

```
@Autowired RandomPhoto randomPhoto;
```

In the retry proxy, there is an imagined `try-catch` block that goes to the core. In our case, the core throws an exception three times. That is, the `catch` block of the retry proxy catches the exception, and the proxy calls the core again.

Let's obtain the image:

```
log.info( "Before receive()" );
String json = randomPhoto.receive( "male" );
log.info( "After receive(): {}", json );
```

No exception appears on the console, and the log indicates that the retry proxy goes to the core several times:

```
… 18:39:58.555  INFO 22222 --- [main] com.….RetryTest : Before receive()
… 18:39:59.000  INFO 22222 --- [main] com.….RandomPhoto : Attempts: 1
… 18:40:00.000  INFO 22222 --- [main] com.….RandomPhoto : Attempts: 2
… 18:40:01.000  INFO 22222 --- [main] com.….RandomPhoto : Attempts: 3
… 18:40:02.000  INFO 22222 --- [main] com.….RetryTest : After receive(): ↩
{"results":[{"picture":{"large":"https://randomuser.me/api/portraits/…
```

The outputs show the three attempts at retrieving the image. The client doesn't notice the repeated calls. After the third attempt, the operation succeeds, and the `receive(…)`

---

[34] *https://docs.spring.io/spring-retry/docs/api/current/org/springframework/retry/annotation/EnableRetry.html*

method in the core outputs a JSON string as a response—unless the service is actually unreachable.

There is an interesting detail that can be read in the log output. It's noticeable from the times that the retry proxy waits 1 second between each method call. Is this a coincidence?

Annotation Type @Retryable

There are several ways to configure `@Retryable` because this annotation type has many annotation attributes:

```
@Target({ ElementType.METHOD, ElementType.TYPE })
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface Retryable {
    String recover() default "";
    String interceptor() default "";
    @AliasFor("include") Class<? extends Throwable>[] retryFor() default{};
    @AliasFor("exclude") Class<? extends Throwable>[] noRetryFor() default{};
    Class<? extends Throwable>[] notRecoverable() default {};
    String label() default "";
    boolean stateful() default false;
    int maxAttempts() default 3;
    String maxAttemptsExpression() default "";
    Backoff backoff() default @Backoff();
    String exceptionExpression() default "";
    String[] listeners() default {};
}
```

Each annotation attribute has a default value, including `maxAttempts`. This attribute determines the number of retry attempts that the retry proxy should make before aborting. The default value for `maxAttempts` is `3`. In the previous example, three retries were attempted because this is the default value. If the number of retries in our example had been set to `4`, the proxy would have thrown an exception.

The number can be set to an `int` via the `maxAttempts` annotation attribute or written as a SpEL expression via `maxAttemptsExpression`.

The annotation type `@Retryable` has an annotation attribute `backoff default @Backoff()` that describes what should happen after an error.

@Backoff: How to Continue after the Error

If the core could not deliver due to an exception, the annotation `@Backoff`[35] tells after which delay it should continue and how. Here, too, there are many setting options:

---

[35] *https://docs.spring.io/spring-retry/docs/api/current/org/springframework/retry/annotation/Backoff.html*

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface Backoff {
    @AliasFor("delay") long value() default 1000;
    @AliasFor("value") long delay() default 1000;
   String delayExpression() default "";
    long maxDelay() default 0;
    String maxDelayExpression() default "";
    double multiplier() default 0;
    String multiplierExpression() default "";
    boolean random() default false;
}
```

The interesting part is a `delay` (alias for `value`), and this specifies a delay of 1,000 milliseconds, or 1 second, by default. We could see this in the log output: when the retry proxy catches the exception, it waits 1 second before going back to the core. The delay can also be written as a SpEL expression.

@Retryable Example with Exceptions and @Backoff

An adjustment of the default values might look like this:

```
@Retryable(
  value       = { TypeOneException.class, TypeTwoException.class },
  maxAttempts = 4,
  backoff     = @Backoff( 2000 /* ms */ )
)
```

It's essential to explicitly mention the exceptions that the retry proxy should handle. In this case, the proxy should respond to the fictional exceptions `TypeOneException` and `TypeTwoException`. Otherwise, the proxy will repeat the code block for every exception, which isn't desirable. For instance, it's unlikely that a `NullPointerException` should trigger code repetition.

4.6.3 Fallback with @Recover

If the call works after as many attempts as defined in `maxAttempts`, there is a happy ending. If the retry proxy fails after the `maxAttempts` number of attempts, a "rescue method" can return a default value. This method is annotated with `Recover`.[36]

In the following, the `@Retryable` method will always fail—it will result in a call to `recover(…)`:

---

[36] *https://docs.spring.io/spring-retry/docs/api/current/org/springframework/retry/annotation/Recover.html*

```
@Retryable
public String read() throws IOException {
  throw new IOException( LocalTime.now().toString() );
}

@Recover
public String recover( IOException e ) {
  System.err.println( "Aarrrrg: " + e );
  return …
}
```

If the method in the core always fails, the method annotated with `@Recover` is called at the very end as a last resort. This method, also called a *fallback* method, receives the exception and returns to the client the result that the `@Retryable` method could not return. The name of the method is irrelevant.

Note

The `@Recover` method must have the identical return type as the `@Retryable` method because the `@Recover` method returns the result that `@Retryable` could not return.

What Does an @Recover Method Return?

An interesting question arises when deciding whether to write methods that conceal the fact that an operation could fail. This can be problematic, especially if remote calls are transparent to the caller. If the caller isn't aware of potential network exceptions, it can lead to unexpected behavior and errors.

While exceptions are a possible solution, it's worth considering other alternatives as well:

- The return type could be an `Optional` or an empty data structure to prevent the receiver from receiving any data in case of a system error.
- Another option is to fall back on cached values. For example, if the program is meant to load a random image but fails, it could retrieve the last successful image from a cache.
- Default values can also be provided to handle failures. In the case of random images, a default value of a URL to an image with ID 0 could be returned.

Ultimately, the decision of how to handle failure depends on the specific use case and requirements of the program.

The solution with the default random image could be programmed in code like this in a `@Recover` method:

```
@Recover
public String recoverFromRandomUserMeConnectionProblems() {
  return """
      {"results":[{"picture":{
      "large":"https://randomuser.me/api/portraits/lego/1.jpg"
      "medium":"https://randomuser.me/api/portraits/med/lego/1.jpg"
      "thumbnail":"https://randomuser.me/api/portraits/thumb/lego/1.jpg"
      }}]}""";
}
```

4.6.4 RetryTemplate

So far, we've explored the declarative approach that employs annotations to generate a proxy in the background. However, it's also possible to implement code repetition on errors without annotations. This approach simplifies the process of repeating arbitrary code blocks without additional steps.

The focus is on the class `RetryTemplate`[37] and the `execute(…)` method that executes the code block. Internally, the retry proxy also works with the `RetryTemplate`.

Schematically, the usage looks like this:

```
RetryTemplate retryTemplate = new RetryTemplate();
retryTemplate.execute( context -> {
    …
    return result;
} );
```

The code block is of type `RetryCallback`[38] and a functional interface:

```
public interface RetryCallback<T> {
  T doWithRetry( RetryContext context ) throws Throwable;
}
```

The framework passes a `RetryContext` object to the method and returns a result. Any exceptions can be thrown—you can't get higher in the exception inheritance hierarchy than `Throwable`.

The `RetryTemplate` class has an overloaded `execute(…)` method that includes an additional parameter for accepting an object of type `RecoverCallback`. This allows options such as the number of retries to be set.

---

[37] *https://docs.spring.io/spring-retry/docs/api/current/org/springframework/retry/support/RetryTemplate.html*

[38] *https://docs.spring.io/spring-retry/docs/api/current/org/springframework/retry/RetryCallback.html*

RetryTemplate Builder

A more user-friendly configuration option is available through the `RetryTemplate` builder. Here is an example of how it can be used:

```
RetryTemplate tryTo = RetryTemplate.builder()
                        .infiniteRetry()
                        .retryOn( IOException.class )
                        .uniformRandomBackoff( 1000 /* ms */, 5000 /* ms */ )
                        .build();
tryTo.execute( context -> {
  return …;
} );
```

The `RetryTemplate` has a static `builder()` method that allows you to parameterize the `RetryTemplate` and build it using `build()`. The fluent API is elegant. For example, `infinityRetry()` can be used to configure an unlimited number of retries, while `retryOn(…)` defines which exceptions the `RetryTemplate` should respond to. `uniformRandomBackoff(…)` configures a random wait time between 1 and 5 seconds.

At the end, there is a `RetryTemplate`, and the `execute(…)` method executes the code block.

4.7 Summary

In the world of wiring, we often encounter proxies that wrap themselves around our objects, rather than instances of our own classes. This chapter has delved into the important proxies in practical use, with a brief touch on Jakarta Bean Validation. Although not the most exciting read, it's worth familiarizing oneself with the comprehensive standard, which can be found at *https://jakarta.ee/specifications/bean-validation/3.0/jakarta-bean-validation-spec-3.0.html*. For a more practical take, the documentation of the reference implementation of Hibernate at *https://docs.jboss.org/hibernate/stable/validator/reference/en-US/html_single/* is recommended.

Caching is another extensive topic that can't be avoided when dealing with caching implementations. Factors such as memory size, usage patterns, and access times become crucial in this area. Furthermore, distributed caching is a fascinating topic that holds significant importance in the cloud environment. As you continue your exploration of these topics, keep an eye out for these crucial aspects.

Chapter 5

Connecting to Relational Databases

In this chapter, we'll explore how Spring applications can interact with relational databases to store, retrieve, and manage data. We'll dive into the various tools and techniques provided by Spring to facilitate database integration, including Java Database Connectivity (JDBC), Spring Data JPA, and more.

5.1 Set Up an H2 Database

Before we dive into our examples, let's make sure we have access to a relational database management system (RDBMS) that provides a database with tables and data. As our application interacts with databases using JDBC, the following examples will be compatible with any database that supports the JDBC interface. For demonstration purposes, we'll use the *H2* (*http://h2database.com*) database management system, which is a compact and easy-to-use solution.

5.1.1 Brief Introduction to the H2 Database

H2 is a database management system that is entirely implemented in Java and open source. Its source code is available for viewing on GitHub (*https://github.com/h2database/h2database*), allowing you to follow its development. The system is licensed under Mozilla Public License Version 2.0 (MPL 2.0) and Eclipse Public License 1.0 (EPL 1.0).

The H2 database management system supports three different modes:

- **Server mode**
  The system runs all day, waits for incoming connections, handles them, and continues to run endlessly. This mode is common for "big" RDBMSs.
- **Embedded mode**
  The system is part of the application, and when the application starts up, H2 starts with a database. The application can read and write the database. If the application shuts down, the embedded database management system is also terminated.
- **In-memory mode**
  In this special form of embedded mode, the database resides exclusively in memory, and H2 doesn't persist any data. Data can be read and written, but everything takes place in main memory. When the program is terminated, all data is lost.

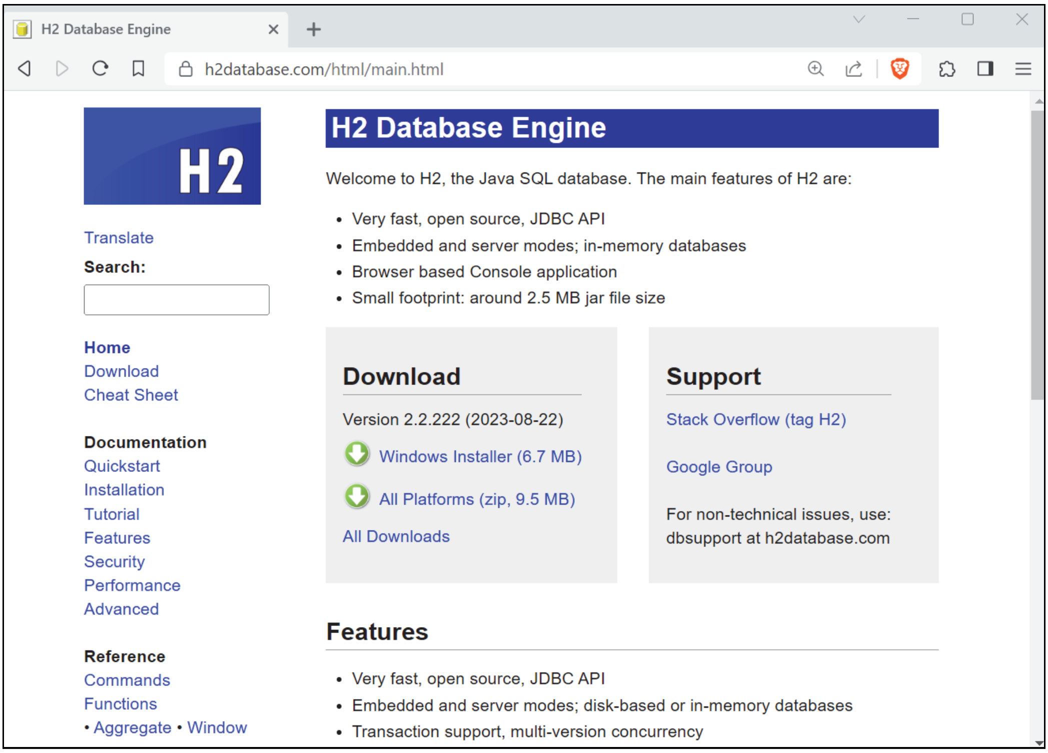
This flexibility makes H2 a versatile option for different use cases, from big data applications to small in-memory databases for testing and development purposes.

Note

Occasionally, in-memory databases are used for tests, but this is only useful if the productive system also uses the same database management system, such as H2. Otherwise, there can be significant discrepancies in SQL syntax and database capabilities when compared to other databases. To avoid these issues, Chapter 7, Section 7.16.6, describes an alternative solution called Project Testcontainer. This approach provides a viable option for testing by creating lightweight, isolated Open Container Initiative (OCI) containers that can be used for database testing.

5.1.2 Install and Launch H2

The next step is to install and start the H2 database locally on the computer. The H2 Database Engine web page (*http://h2database.com*) shows the different download options (see Figure 5.1).


H2 Database Engine
h2database.com/html/main.html
H2
Translate
Search:
Home
Download
Cheat Sheet
Documentation
Quickstart
Installation
Tutorial
Features
Security
Performance
Advanced
Reference
Commands
Functions
• Aggregate • Window

H2 Database Engine

Welcome to H2, the Java SQL database. The main features of H2 are:

- Very fast, open source, JDBC API
- Embedded and server modes; in-memory databases
- Browser based Console application
- Small footprint: around 2.5 MB jar file size

Download

Version 2.2.222 (2023-08-22)

Windows Installer (6.7 MB)

All Platforms (zip, 9.5 MB)

All Downloads

Support

Stack Overflow (tag H2)

Google Group

For non-technical issues, use:
dbsupport at h2database.com

Features

- Very fast, open source, JDBC API
- Embedded and server modes; disk-based or in-memory databases
- Transaction support, multi-version concurrency

Figure 5.1 H2 Download Options

There are several ways to get H2 running: via a **Windows Installer**, as a ZIP archive for **All Platforms**, or by downloading the old versions under **All Downloads**.

Click on **All Platforms**, download the 10 mebibyte (MiB) ZIP archive, and extract its contents. This will create a directory named *h2* with the following files:

- *bin*
- *docs*
- *src*
- *build.bat*
- *build.iml*
- *build.sh*

The *bin* subdirectory contains a runnable Java archive and batch files for Windows and Unix systems:

- *h2-2.2.222.jar*
- *h2.bat*
- *h2.sh*
- *h2w.bat*

Those on the graphical interface can try double-clicking on the JAR file. That should start H2, provided the `.jar` file extension has the Java virtual machine (JVM) associated with it. Alternatively, it can be started from the shell files or from the command line with `java -jar h2-2.2.222.jar` (on Windows also `javaw -jar h2-2.2.222.jar`).

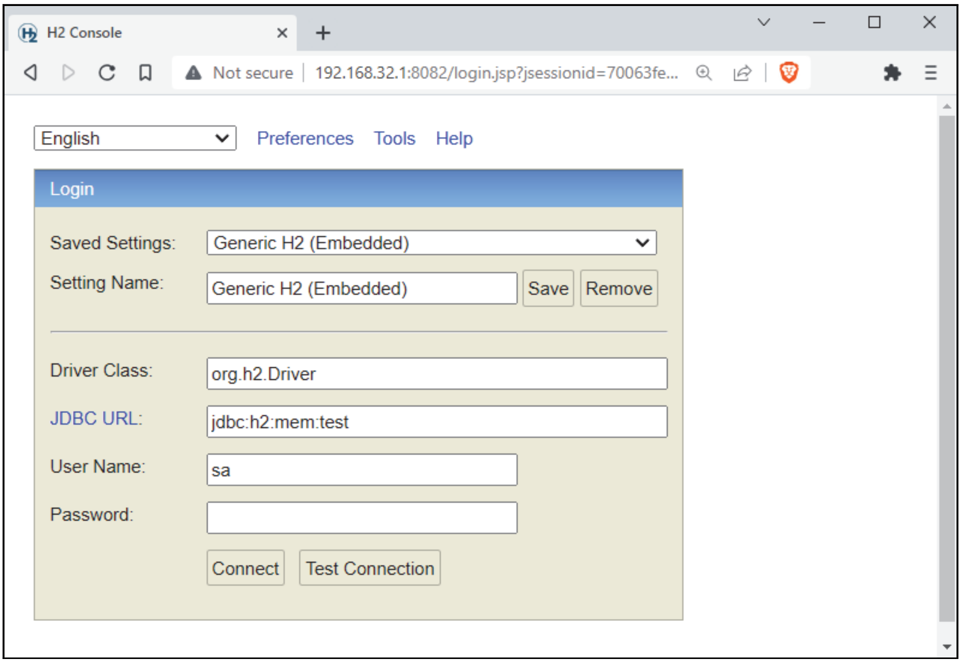
Note

To run the examples, the H2 database server must always be running. However, after each system reboot, H2 needs to be manually restarted. To simplify this process, H2 can be set up as a service to automatically start the RDBMS upon system restart. This ensures that the H2 database is always available for use without requiring manual intervention.

When H2 starts, a GUI opens automatically in the web browser.

Tip: Troubleshooting Tips

If the web interface shows an error, the first suggested solution is to use the URL *http://localhost:8082/*. Because H2 uses port 8082 by default for the web interface, but the port may already be occupied by another application, H2 can use a different port. To archive this, modify the *.h2.server.properties* file in the user directory (or create a new one), and enter the following, for example: “webPort=8888”.


H2 Console
Not secure
192.168.32.1:8082/login.jsp?jsessionid=70063fe...
English
Preferences
Tools
Help
Login
Save
Remove
Connect
Test Connection

Saved Settings: Generic H2 (Embedded)
Setting Name: Generic H2 (Embedded)

Driver Class: org.h2.Driver
JDBC URL: jdbc:h2:mem:test
User Name: sa
Password:

Figure 5.2 H2 Console in the Web Browser

Create a New Database via the GUI

The H2 Console is a small admin interface so that you can connect to the database. However, we don't have a database yet, so we need to create one because it's not created on first access.[1] This can be done via the GUI or via the shell.

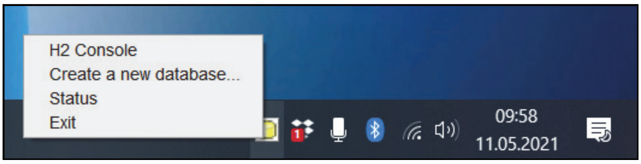

H2 Console
Create a new database...
Status
Exit
09:58
11.05.2021

Figure 5.3 H2 Icon in the System Tray

In Windows operating systems, there is a system tray located at the bottom right of the screen. This tray contains various icons, some of which may be hidden. When H2 is running, a small yellow barrel icon can be seen in the system tray. For macOS users, the H2 barrel icon can be found in the upper area of the screen. Keeping an eye on this icon is useful for quickly checking the status of H2 and ensuring that it's running properly.

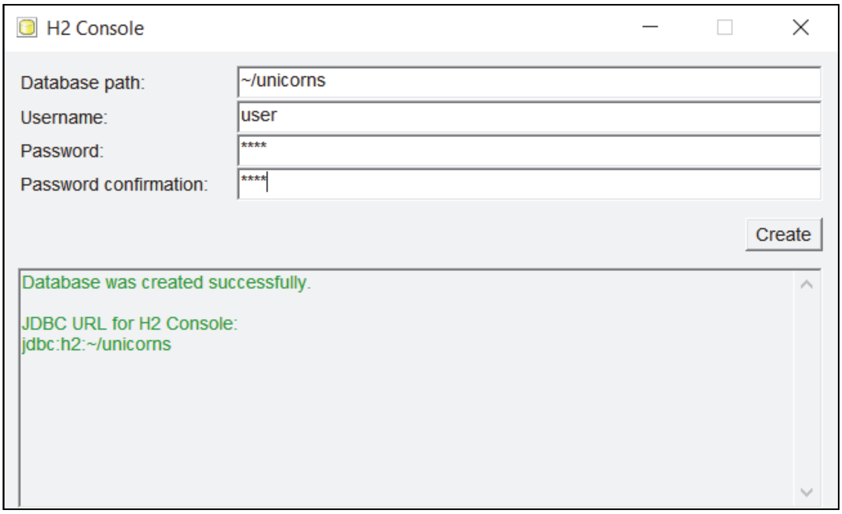
---

[1] In early versions of H2, a database was indeed created on first access, but this has long been turned off.

Tip

Once you've closed the H2 Console, clicking on the barrel takes you to a new tab with the web interface.

An activation of the context menu on the barrel shows four options (see Figure 5.3). We need one of them now because we need to create a new database, so we click on **Create a new database**. A new dialog follows, as shown in Figure 5.4.



Database path: ~/unicorns
Username: user
Password: ****
Password confirmation: ****

```
Database was created successfully.

JDBC URL for H2 Console:
jdbc:h2:~/unicorns
```

Figure 5.4 Dialog for Creating a New Database

Using the dialog, you can create a new database on the file system. However, when performing a new installation, it's not recommended to use the default database path `./test`. This is because using this path would create a new database with the name `test` in the directory where H2 was started. This can cause confusion and potential issues, so it's recommended to choose a more specific and appropriate path for the new database.

Enter the following in the text fields of the dialog:

- **Database path**: "~/unicorns"
- **Username**: "user
- **Password**: "pass"
- **Password confirmation**: "pass"

To create the database in the user directory, enter "~/unicorns" in the dialog box. You can select any username and password you like, but for simplicity, let's keep them easy to remember. Click on **Create** to create the database. Once the database is created, a green message will appear in the lower text field. The actual database file, *unicorns.mv.db*, will be located in the user directory.

Create a New Database via the Shell *

A database can also be created via the command line:

```
$ java -cp h2-2.2.222.jar org.h2.tools.Shell

Welcome to H2 Shell 2.2.222 (2023-08-22)
Exit with Ctrl+C
[Enter]   jdbc:h2:~/test
URL       jdbc:h2:~/unicorns
[Enter]   org.h2.Driver
Driver
[Enter]
User      user
Password
Connected
Commands are case insensitive; SQL statements end with ';'
help or ?      Display this help
list           Toggle result list / stack trace mode
maxwidth       Set maximum column width (default is 100)
autocommit     Enable or disable autocommit
history        Show the last 20 statements
quit or exit   Close the connection and exit

sql> exit
Connection closed
```

The program is an interactive shell that can be used to create databases. SQL statements can also be executed via the shell.

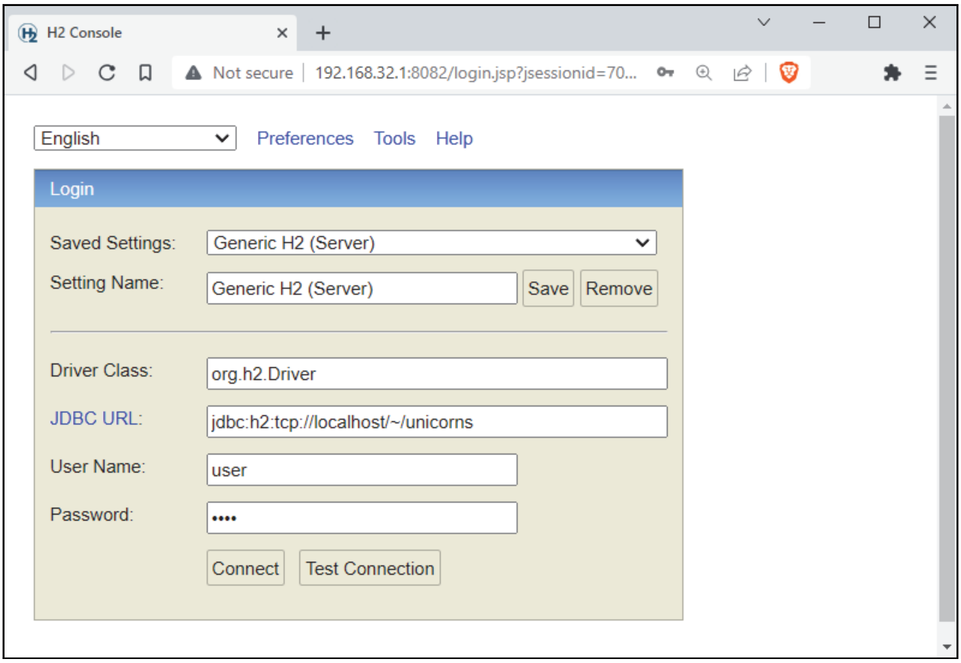
5.1.3 Connect to the Database via the H2 Console

After the start, the H2 Console is displayed (see Figure 5.5), and a connection to the new database can be established. (By the way, the H2 Console can connect to different databases, not just H2.)

We entered the following:

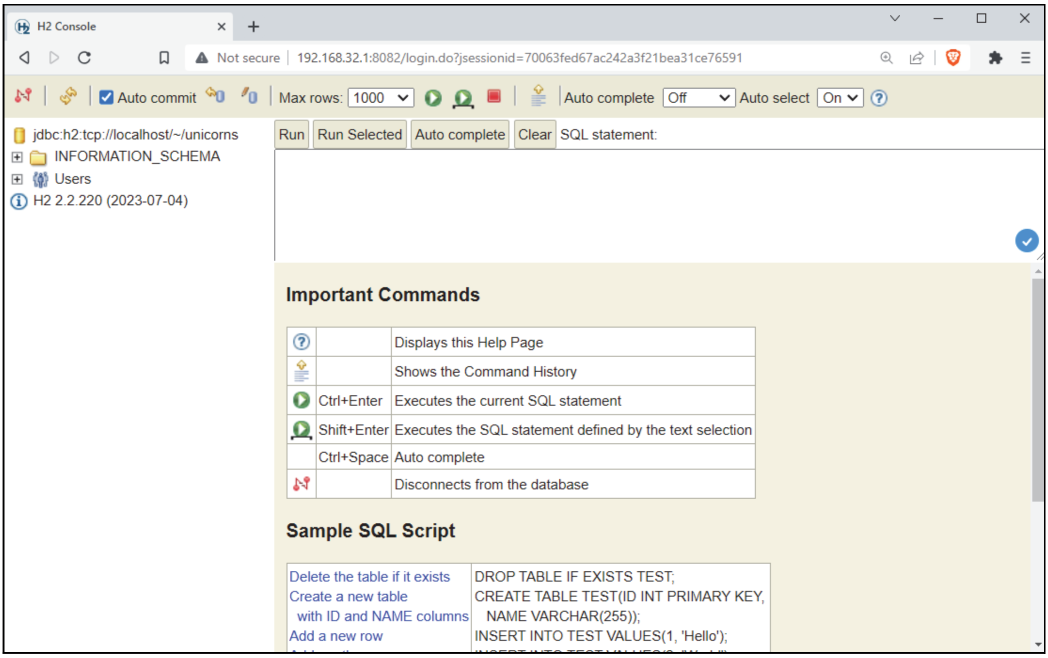
- **JDBC URL**: “jdbc:h2:tcp://localhost/~/unicorns”
- **Username**: “user”
- **Password**: “pass”

Clicking **Connect** will take us to a new page where tables can be viewed and SQL statements can be issued (see Figure 5.6).


H2 Console
Not secure
192.168.32.1:8082/login.jsp?jsessionid=70...
English
Preferences
Tools
Help
Login
Save
Remove
Connect
Test Connection

Saved Settings: Generic H2 (Server)
Setting Name: Generic H2 (Server)
Driver Class: org.h2.Driver
JDBC URL: jdbc:h2:tcp://localhost/~/unicorns
User Name: user
Password: ••••

Figure 5.5 Enter Connection Data


H2 Console
Not secure
192.168.32.1:8082/login.do?jsessionid=70063fed67ac242a3f21bea31ce76591
Auto complete
Auto select
Run
Run Selected
Auto complete
Clear
SQL statement:
jdbc:h2:tcp://localhost/~/unicorns
INFORMATION_SCHEMA
Users
H2 2.2.220 (2023-07-04)

☑ Auto commit
Max rows: 1000
Auto complete: Off
Auto select: On

Important Commands

?		Displays this Help Page
		Shows the Command History
	Ctrl+Enter	Executes the current SQL statement
	Shift+Enter	Executes the SQL statement defined by the text selection
	Ctrl+Space	Auto complete
		Disconnects from the database

Sample SQL Script

Delete the table if it exists	DROP TABLE IF EXISTS TEST;
Create a new table with ID and NAME columns	CREATE TABLE TEST(ID INT PRIMARY KEY, NAME VARCHAR(255));
Add a new row	INSERT INTO TEST VALUES(1, 'Hello');

Figure 5.6 The H2 Console with Its SQL Editor

5.1.4 Date4u Database Schema

For the database to store information from the Date4u application, the four tables from Figure 5.7 are used.

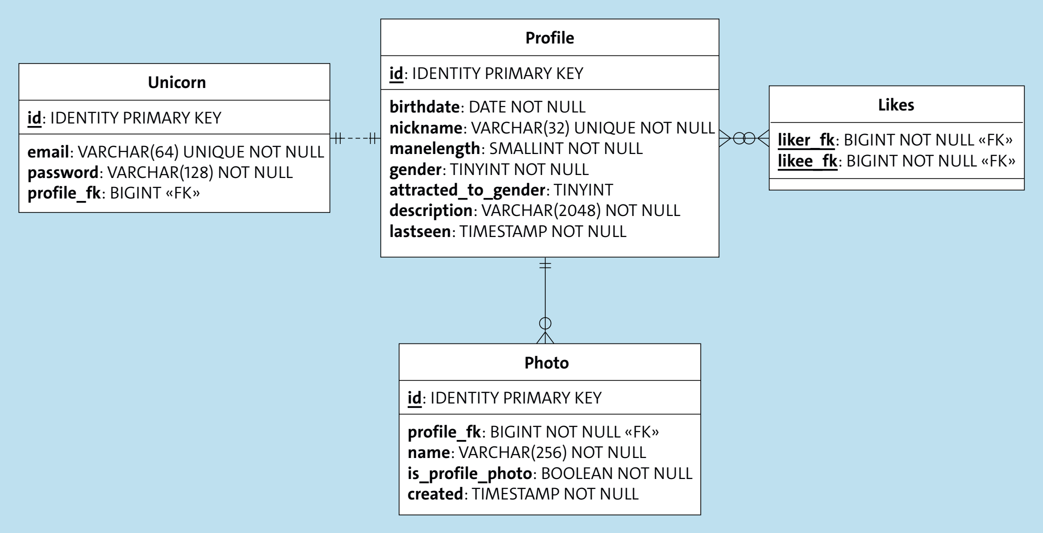

Unicorn
id: IDENTITY PRIMARY KEY
email: VARCHAR(64) UNIQUE NOT NULL
password: VARCHAR(128) NOT NULL
profile_fk: BIGINT «FK»
Profile
id: IDENTITY PRIMARY KEY
birthdate: DATE NOT NULL
nickname: VARCHAR(32) UNIQUE NOT NULL
manelength: SMALLINT NOT NULL
gender: TINYINT NOT NULL
attracted_to_gender: TINYINT
description: VARCHAR(2048) NOT NULL
lastseen: TIMESTAMP NOT NULL
Likes
liker_fk: BIGINT NOT NULL «FK»
likee_fk: BIGINT NOT NULL «FK»
Photo
id: IDENTITY PRIMARY KEY
profile_fk: BIGINT NOT NULL «FK»
name: VARCHAR(256) NOT NULL
is_profile_photo: BOOLEAN NOT NULL
created: TIMESTAMP NOT NULL

Figure 5.7 The Schema of the Date4u Database

The four tables have the following properties:

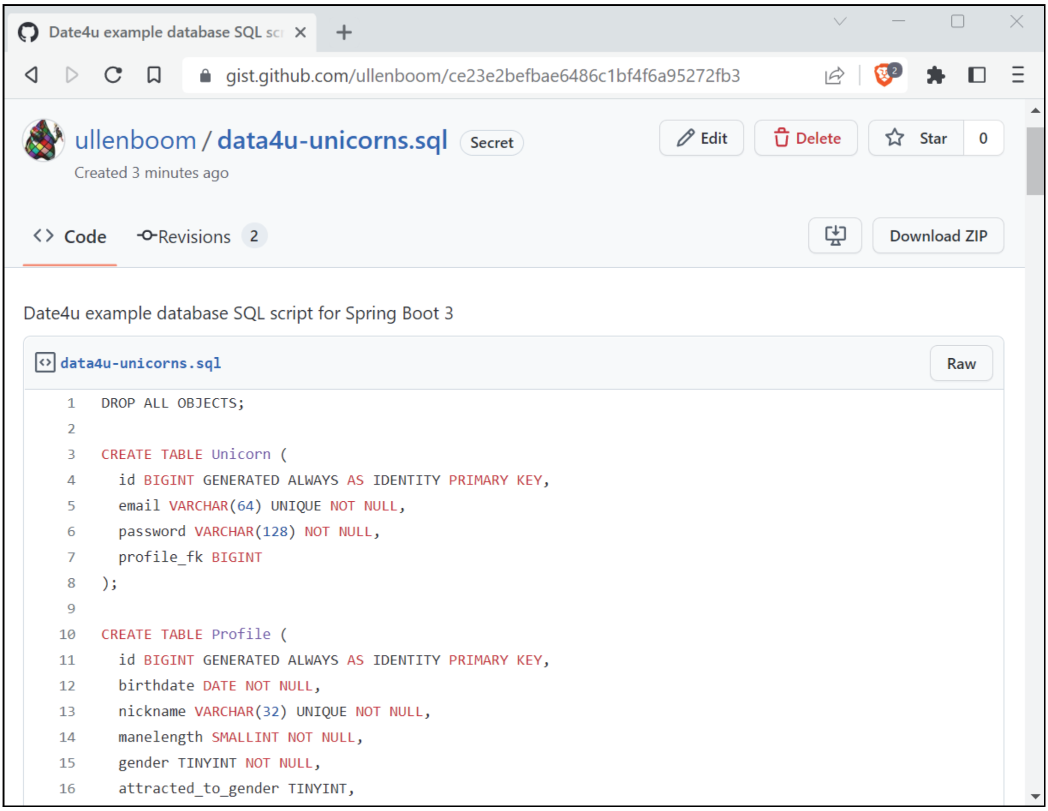
- **Profiles**
  At the center is the `Profiles` table, which stores the profile of a unicorn. Each profile has an `id` that is automatically generated using an identity column. The profile of the unicorn has a `birthdate` that must be given: it's `NOT NULL`. The unicorn has a `nickname`, which is limited to 32 characters. It must be `UNIQUE` and must also be `NOT NULL`. The mane length (`manelength`) is of type `SMALLINT` and must be `NOT NULL`. The `gender` must also be `NOT NULL`. `attracted_to_gender` stores which gender the unicorn is attracted to; it's a `TINYINT` and can be `NULL` to express that the unicorn is interested in any other gender. Each profile has a `description` with a maximum of 2,048 characters. The `lastseen` column stores when the unicorn was last seen as a date-time stamp, and this is also `NOT NULL`.
- **Unicorn**
  Each unicorn is represented in the `Unicorn` table. Each unicorn has an `id` (which in turn is an identity column), an `email` address, a (hashed) `password`, and a reference to the corresponding profile in `profile_fk`. The direction of the 1:1 relationship is didactically motivated; of course, the profile can also reference the unicorn. This would have the advantage that unicorns can also be administrators.
- **Photo**
  A profile can reference any number of photos in a 1:n relationship. Each photo has an

`id` (again, an identity column). There is a reference to the corresponding profile via a foreign key relationship in the `profile_fk` column. Each photo has a name. Later, the `PhotoService` can load an image from the file system for the image name. Whether a photo is a profile photo is determined by `is_profile_photo`, whereby the software must ensure that a profile photo can only be set once per profile. The timestamp `created` indicates when the profile photo was uploaded. All columns must be `NOT NULL`.

- **Likes**
  Join table `Likes` implements an n:m relationship between the profiles that like each other. The column type is `BIGINT` and `NOT NULL`. The relationship is directed.

Fill the Date4u Demo Database with Data

The database with demo data can be obtained from *https://tinyurl.com/4fu3hwu4* as an SQL script (see Figure 5.8).

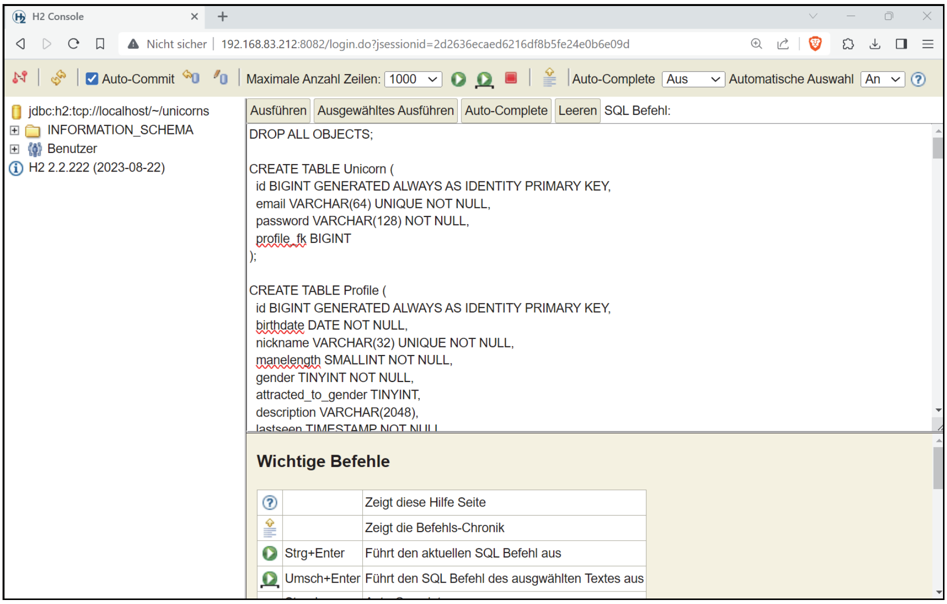

Date4u example database SQL sc
gist.github.com/ullenboom/ce23e2befbae6486c1bf4f6a95272fb3
ullenboom / data4u-unicorns.sql
Secret
Created 3 minutes ago
Edit
Delete
Star
0
Code
Revisions 2
Download ZIP
data4u-unicorns.sql
Raw

Date4u example database SQL script for Spring Boot 3

```sql
DROP ALL OBJECTS;

CREATE TABLE Unicorn (
  id BIGINT GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
  email VARCHAR(64) UNIQUE NOT NULL,
  password VARCHAR(128) NOT NULL,
  profile_fk BIGINT
);

CREATE TABLE Profile (
  id BIGINT GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
  birthdate DATE NOT NULL,
  nickname VARCHAR(32) UNIQUE NOT NULL,
  manelength SMALLINT NOT NULL,
  gender TINYINT NOT NULL,
  attracted_to_gender TINYINT,
```

Figure 5.8 SQL Script with Schema Definition and Demo Data

The SQL statements can be placed in the text field of the H2 Console and sent by clicking **Execute**. This way, the database is filled (see Figure 5.9).


H2 Console
Nicht sicher | 192.168.83.212:8082/login.do?jsessionid=2d2636ecaed6216df8b5fe24e0b6e09d
☑ Auto-Commit
Maximale Anzahl Zeilen: 1000
Auto-Complete Aus
Automatische Auswahl An
jdbc:h2:tcp://localhost/~/unicorns
INFORMATION_SCHEMA
Benutzer
H2 2.2.222 (2023-08-22)
Ausführen
Ausgewähltes Ausführen
Auto-Complete
Leeren
SQL Befehl:

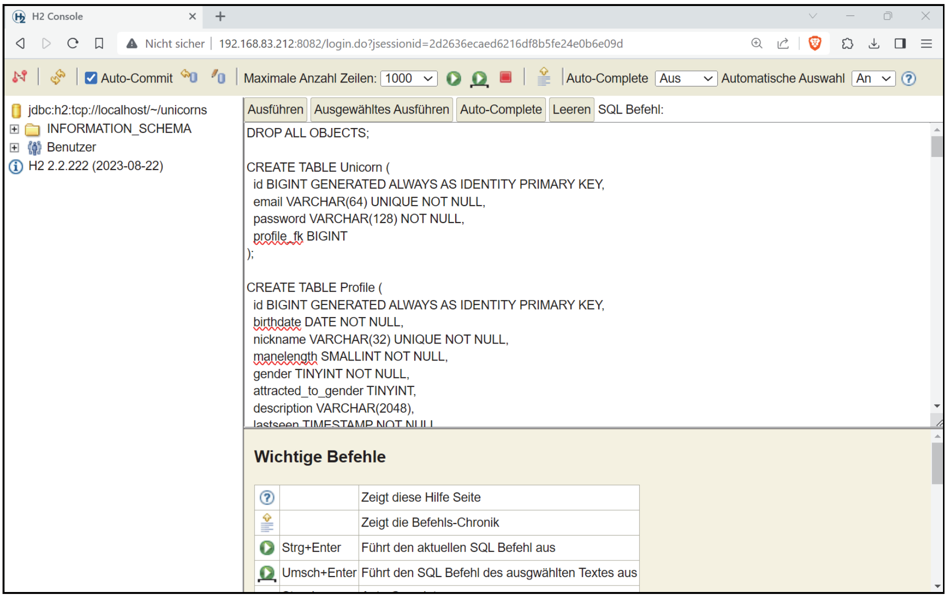
```
DROP ALL OBJECTS;

CREATE TABLE Unicorn (
  id BIGINT GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
  email VARCHAR(64) UNIQUE NOT NULL,
  password VARCHAR(128) NOT NULL,
  profile_fk BIGINT
);

CREATE TABLE Profile (
  id BIGINT GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
  birthdate DATE NOT NULL,
  nickname VARCHAR(32) UNIQUE NOT NULL,
  manelength SMALLINT NOT NULL,
  gender TINYINT NOT NULL,
  attracted_to_gender TINYINT,
  description VARCHAR(2048),
  lastseen TIMESTAMP NOT NULL
```

Wichtige Befehle

?		Zeigt diese Hilfe Seite
		Zeigt die Befehls-Chronik
▶	Strg+Enter	Führt den aktuellen SQL Befehl aus
▶	Umsch+Enter	Führt den SQL Befehl des ausgewählten Textes aus

Figure 5.9 Accepting an SQL Script in the H2 Console

Subsequently, four new tables appear on the left side (see Figure 5.10). A click on **Clear** clears the input field. If you click on one of the tables on the left, a `SELECT` statement is automatically generated, which **Execute** processes again.


H2 Console
Nicht sicher | 192.168.83.212:8082/login.do?jsessionid=2d2636ecaed6216df8b5fe24e0b6e09d
☑ Auto-Commit
Maximale Anzahl Zeilen: 1000
Auto-Complete Aus
Automatische Auswahl An
jdbc:h2:tcp://localhost/~/unicorns
INFORMATION_SCHEMA
Benutzer
H2 2.2.222 (2023-08-22)
Ausführen
Ausgewähltes Ausführen
Auto-Complete
Leeren
SQL Befehl:

```
DROP ALL OBJECTS;

CREATE TABLE Unicorn (
  id BIGINT GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
  email VARCHAR(64) UNIQUE NOT NULL,
  password VARCHAR(128) NOT NULL,
  profile_fk BIGINT
);

CREATE TABLE Profile (
  id BIGINT GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
  birthdate DATE NOT NULL,
  nickname VARCHAR(32) UNIQUE NOT NULL,
  manelength SMALLINT NOT NULL,
  gender TINYINT NOT NULL,
  attracted_to_gender TINYINT,
  description VARCHAR(2048),
  lastseen TIMESTAMP NOT NULL
```

Wichtige Befehle

?		Zeigt diese Hilfe Seite
		Zeigt die Befehls-Chronik
▶	Strg+Enter	Führt den aktuellen SQL Befehl aus
▶	Umsch+Enter	Führt den SQL Befehl des ausgewählten Textes aus

Figure 5.10 SQL Editor with Result

By the way, it may well be that the table contents look a bit different by now.

Note: Naming Convention

To improve readability and distinguish SQL statements, a convention is often used where SQL keywords such as `SELECT`, `FROM`, and `WHERE` are written in all capital letters. Tables, on the other hand, are written with the first letter capitalized, such as `Profiles`, while columns are written in lowercase, such as `id`. This convention not only helps distinguish different parts of a SQL statement but also makes the code easier to read and understand for other developers who may work on the same project.

5.2 Realize Database Accesses with Spring

Inevitably, most applications will require external data storage at some point. These are commonly relational databases, although NoSQL databases are also used for tasks such as document storage or text search. Additionally, external documents may be provided in formats such as XML, Excel, or comma-separated values (CSV) and will need to be read and written by the application.

5.2.1 Spring Helper

Spring provides a range of abstractions to access various database management systems and data formats. At the lowest level, it offers simplified access to the JDBC connection. As we move up the stack, SQL is abstracted away to a great extent. At a certain point, it becomes impossible to tell from the code whether the data is being managed by relational or NoSQL databases.

At a rather low level of abstraction, Spring offers the following:

- If you access relational databases, this is usually done with JDBC.[2] A JDBC connection is established via `java.sql.DriverManager` or `javax.sql.DataSource`. The Spring Framework builds a `DataSource` directly if certain configuration properties are set; no access via the `DriverManager` is necessary in the code.
- Somewhat more abstracted from the JDBC data types `Connection`, `Statement`, and `ResultSet` are the classes `[NamedParameter]JdbcTemplate`. These classes can easily be used to send typical queries to the relational database. These template classes are typical facades. That is, they still work with JDBC in the background, but typical queries can be written as one-liners. Template classes exist not only for JDBC access but also for other database management systems, for example, the `MongoTemplate` for the MongoDB database and many other template classes.
- Spring translates exceptions from the data stores into its own exception hierarchy, with the unchecked `DataAccessException` being the base type. To complete this,

---

[2] There is now actually an alternative to JDBC, and that is Reactive Relational Database Connectivity (R2DBC; *https://r2dbc.io*). So far, however, this alternative is still very new and also proprietary.

Spring internally uses translation tables that map specific exceptions and error codes from a database to Spring exceptions.

- In object-oriented software development, we shouldn't think in terms of tables and columns, but only in terms of objects. It's important to map the table structure to objects, that is, *entity beans*. Of course, code can map columns to objects, but this work is done more comfortably by *object-relational mapping* (O/R mappers for short). The Jakarta family defines a standard called the *Jakarta Persistence API*. It in turn declares the `EntityManager` type, which Spring can build automatically. With a wired `EntityManager`, an application can store objects in the database via appropriate methods and also fetch objects from the database via queries.

All these data types are rather low level because they are still closely related to the native APIs (e.g., JDBC) or the Jakarta Persistence API.

The *Spring Data* project operates at a higher level of abstraction, focusing on facilitating the implementation of repositories. A repository is situated in the data access layer and offers an interface for accessing the data store.

Spring Data provides interfaces for performing *create, read, update, delete (CRUD) operations*. What sets Spring Data apart is its ability to automatically generate CRUD repositories for various database management systems. Essentially, by making a small code adjustment, it's possible to swap out the entire database system.

In the following sections and also in Chapter 6, we'll go deeper on Spring JDBC and then into detail about the Spring Data project. As members of the Spring Data family, the focus is on Spring Data *JPA*, *Spring Data MongoDB*, and *Spring Data Elasticsearch*.

5.3 Spring Boot Starter JDBC

The Spring Framework provides helper classes, such as `JdbcTemplate`. A special starter provides auto-configuration and includes `spring-jdbc` with the helper classes.

5.3.1 Include the JDBC Starter in the Project Object Model

The starter can be included in the project object model (POM) as usual:

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-jdbc</artifactId>
</dependency>
```

Listing 5.1 pom.xml

This *Spring Boot Starter JDBC* is an extension of the core starter. This means that if you turn the Spring Boot Starter into a Spring Boot Starter JDBC project, the core starter will still be included. This is shown by the dependencies in Figure 5.11.

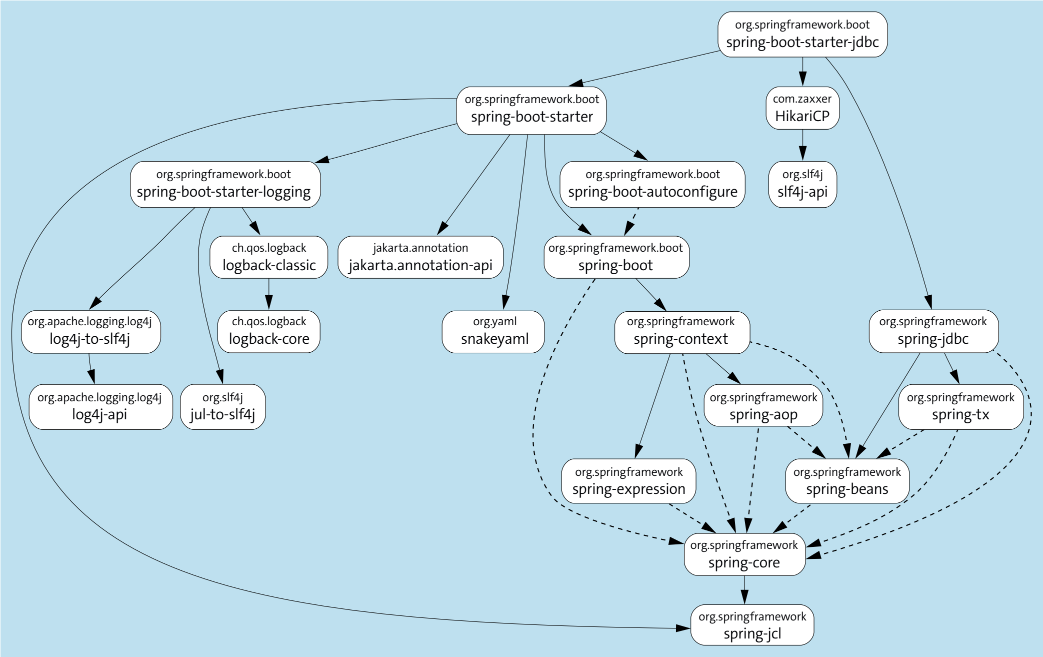

org.springframework.boot
spring-boot-starter-jdbc
org.springframework.boot
spring-boot-starter
com.zaxxer
HikariCP
org.springframework.boot
spring-boot-starter-logging
org.springframework.boot
spring-boot-autoconfigure
org.slf4j
slf4j-api
ch.qos.logback
logback-classic
jakarta.annotation
jakarta.annotation-api
org.springframework.boot
spring-boot
org.apache.logging.log4j
log4j-to-slf4j
ch.qos.logback
logback-core
org.yaml
snakeyaml
org.springframework
spring-context
org.springframework
spring-jdbc
org.apache.logging.log4j
log4j-api
org.slf4j
jul-to-slf4j
org.springframework
spring-aop
org.springframework
spring-tx
org.springframework
spring-expression
org.springframework
spring-beans
org.springframework
spring-core
org.springframework
spring-jcl

Figure 5.11 Spring Boot Starter JDBC Dependencies

When database connection information is available, auto-configuration ensures that objects (e.g., a `DataSource` or the `JdbcTemplate`) are preconfigured and that connection pooling is initialized.

Database JDBC Driver in the Classpath

Every Java program needs a JDBC driver in the classpath to access a database. For H2, it's the dependency.

```
<dependency>
  <groupId>com.h2database</groupId>
  <artifactId>h2</artifactId>
  <scope>runtime</scope>
</dependency>
```

Listing 5.2 pom.xml Extension

Spring Boot takes care of managing the version of H2, but it's only necessary to specify it if version compatibility is a concern. While the H2 version number isn't as important when using it as a pure JDBC client, it becomes crucial when using H2 in production and for testing purposes. Therefore, it's recommended to ensure that the version number is consistent across all instances where H2 is used.

5.3.2 Provide JDBC Connection Data

Next, the connection data must be set. For example, the connection URL and the username and password go into the *application.properties* file.

```
spring.datasource.url=jdbc:h2:tcp://localhost/~/unicorns
spring.datasource.username=user
spring.datasource.password=pass
```

Listing 5.3 application.properties Extension

If Spring Boot finds set property variables `spring.datasource.url`, `spring.datasource.username`, and `spring.datasource.password`, a `DataSource` is built directly thanks to auto-configuration.

5.3.3 Inject DataSource or JdbcTemplate

The interface `DataSource`[3] is a part of Java SE, and it allows for database connections to be obtained through its implementations, which are typically provided by the container. This is a crucial concept because, by using the `DataSource` interface, the client can obtain a connection to the database without having to provide specific database connection information, such as a username and password. We'll just call `getConnection()` through the `DataSource`, and that gives a connection to the database.

It's worth noting that this information doesn't typically belong in a regular program. The `DataSource` is considered a *container-managed object*, meaning that the environment builds and manages the object. If using Spring, the Spring container is responsible for constructing and preparing the `DataSource` object.

A client can have the `DataSource` injected and establish a connection, something like this:

```
@Autowired
DataSource dataSource;
```

The `DataSource` interface doesn't offer as much (see Figure 5.12); the most important is `getConnection(…)`.

We won't stay at this low level of abstraction, but get practically into database queries with the `JdbcTemplate`.

---

[3] *https://docs.oracle.com/en/java/javase/17/docs/api/java.sql/javax/sql/DataSource.html*

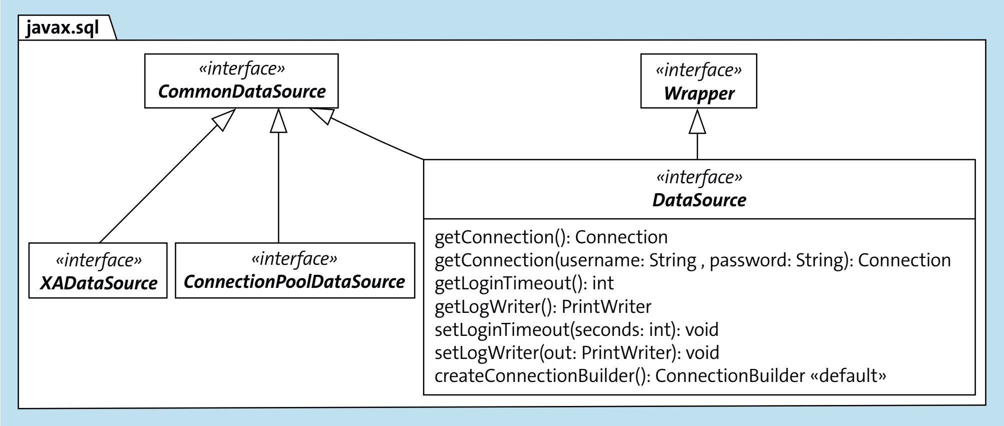

javax.sql
«interface»
CommonDataSource
«interface»
Wrapper
«interface»
DataSource
getConnection(): Connection
getConnection(username: String , password: String): Connection
getLoginTimeout(): int
getLogWriter(): PrintWriter
setLoginTimeout(seconds: int): void
setLogWriter(out: PrintWriter): void
createConnectionBuilder(): ConnectionBuilder «default»
«interface»
XADataSource
«interface»
ConnectionPoolDataSource

Figure 5.12 Type Relationship and Methods of DataSource

Inject JdbcTemplate

Due to set configuration parameters, the auto-configuration leads to a Spring-managed bean of type `JdbcTemplate`, which is pre-initialized with the also built `DataSource`. An injection is therefore possible:

```
@Autowired
JdbcTemplate jdbcTemplate;
```

The constructor of `JdbcTemplate` can also be used to assign another `DataSource`.

Here's an example of using `JdbcTemplate`: What is the mane length of a profile if you want to use the nickname for the search?

```
String sql = "SELECT manelength FROM Profile WHERE nickname = ?";
List<Integer> lengths = jdbcTemplate.queryForList( sql, Integer.class,
                                                   "FillmoreFat" );
lengths.stream().findFirst()
    .ifPresent( len -> log.info( "Mane length: {}", len ) );
```

There are several ways to get this information via the `JdbcTemplate`. The program uses the `queryForList(…)` method, and three pieces of information are passed:

- The SQL string
- The type token for the return (the mane length is an `Integer`)
- The question mark, which is a part of a prepared statement and is filled here by the nickname `FillmoreFat`

The result is a list of numbers. There are two expected returns:

- The list could be empty because there is no profile for this nickname.
- The list contains a single entry.

(If a nickname is found in the database, only one profile with that name can exist because they are always `UNIQUE` according to the schema. Therefore, there can't be two profiles with the same nickname.)

The logic behind the stream expression is to output the length of an element if it exists, and do nothing if it doesn't. While an `if` statement could have been shorter, it's a different matter altogether.

Task: New Shell Component with Database Query

In the following task, we use the shell to produce interactive queries via these steps:

1. Create a new class `JdbcCommands` annotated with `@ShellComponent`.
2. Inject a `JdbcTemplate`.
3. Write a new command `manelength` that determines the mane length of a unicorn based on its nickname. For example, the following can be called from the shell:

   ```
   manelength FillmoreFat
   ```

4. Consider what the output should be if there is no profile with that name.

The proposed solution is in the following listing.

```
@ShellComponent
public class JdbcCommands {
  private final JdbcTemplate jdbcTemplate;

  public JdbcCommands( JdbcTemplate jdbcTemplate ) {
    this.jdbcTemplate = jdbcTemplate;
  }

  @ShellMethod( "Display mane length of a given profile by nickname" )
  public String manelength( String nickname ) {

    String sql = "SELECT manelength FROM Profile WHERE nickname = ?";
    List<Integer> lengths = jdbcTemplate.queryForList( sql, Integer.class,
                                                       nickname );

    return lengths.isEmpty() ? "Unknown profile for nickname " + nickname
                             : lengths.get( 0 ).toString();
  }
}
```

Listing 5.4 JdbcCommands.java

5.3.4 Connection Pooling

The Spring Boot Starter JDBC or extensions such as Spring Boot Starter Data JPA provide a set of preconfigured libraries for accessing and manipulating relational databases. However, the Spring Boot Starter JDBC not only references `spring-jdbc` but also incorporates *HikariCP* (*https://github.com/brettwooldridge/HikariCP*). HikariCP is an open-source *connection pooling* library that enhances the performance of database access by managing and reusing database connections.

When a Java application interacts with a database, it establishes a connection to the database through JDBC. The creation and initialization of a connection can be an expensive process that can consume significant time and resources. Connection pooling helps to alleviate this performance overhead by establishing a pool of pre-initialized database connections that can be reused by multiple application threads.

In the log, it's clear that HikariCP is used:

```
… INFO 20176 --- [           main] ↩
com.zaxxer.hikari.HikariDataSource     : HikariPool-1 - Starting...
… INFO 20176 --- [           main] ↩
com.zaxxer.hikari.HikariDataSource     : HikariPool-1 - Start completed.
…
… INFO 20176 --- [ionShutdownHook] ↩
com.zaxxer.hikari.HikariDataSource     : HikariPool-1 - Shutdown initiated...
… INFO 20176 --- [ionShutdownHook] ↩
com.zaxxer.hikari.HikariDataSource     : HikariPool-1 - Shutdown completed.
```

So, the `JdbcTemplate` doesn't talk directly to the H2 database, but HikariCP hangs in between. If you inject a `DataSource` as

```
@Autowired DataSource dataSource;
```

and then query the `Class object`, the name would look like this:

```
log.info( dataSource.getConnection().getClass().getName() );
// com.zaxxer.hikari.pool.HikariProxyConnection
```

When HikariCP is used with `spring-jdbc`, it installs a proxy around the actual JDBC connection. This proxy intercepts the `close()` method call on a connection, and instead of immediately closing the connection, it keeps the connection open for a period of time. This feature allows the connection to be reused, which reduces the overhead of creating a new connection and improves the performance of the application.

Different Pooled Connection Implementations

There are quite a few connection pooling libraries in the Java universe. HikariCP has excellent performance properties, which is why the Spring Boot team chose these libraries. Spring Boot evaluates dozens of configuration properties; the reference documentation lists them at *https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#appendix.application-properties.data*.

In principle, other implementations can be used, and auto-configuration supports other connection pool implementations; details can be found at *https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#data.sql.datasource.connection-pool*.

5.3.5 Log JDBC Accesses

The connection proxy is essential for performance because establishing new connections is expensive. Another interesting proxy worth noting is `P6Spy`. P6Spy is a JDBC logger that has been in use for nearly two decades, with version 1.0 released in 2005. P6Spy is an open-source library (*https://github.com/p6spy/p6spy*) that can be used to intercept and log database interactions, including SQL statements, parameter values, and execution times. By using P6Spy, developers can gain insights into the performance of their database queries and identify potential bottlenecks or optimizations. P6Spy works by intercepting JDBC method calls using a proxy. When an application makes a JDBC call, the proxy intercepts it and logs the relevant information before passing the call on to the underlying driver. This way, P6Spy can log all database interactions transparently, without requiring any changes to the application code.

P6Spy doesn't need to be prepared in code manually for Spring Boot applications. This is done by the *Spring Boot DataSource Decorator* project (*https://github.com/gavlyukovskiy/spring-boot-data-source-decorator*), which auto-configures P6Spy and other pooled connection implementations. The following must be entered in the POM file for this purpose:

```
<dependency>
  <groupId>com.github.gavlyukovskiy</groupId>
  <artifactId>p6spy-spring-boot-starter</artifactId>
  <version>1.9.0</version>
</dependency>
```

If you start an application, the flow is no longer `JdbcTemplate` → HikariCP → H2-Connection, but `JdbcTemplate` → P6Spy → HikariCP → H2-Connection.

There are new messages in the log stream, such as the following:

```
… --- [main] p6spy : #1632839285730 | ↩
took 3ms | statement | connection 8| url jdbc:h2:tcp://localhost/~/unicorns
SELECT manelength FROM Profile WHERE nickname = ?
SELECT manelength FROM Profile WHERE nickname = 'FillmoreFat';
```

The request made took only 3 milliseconds to complete. However, it's worth noting that there are two outputs due to the use of prepared statements. The first output is the original prepared statement, while the second output displays the same prepared statement with placeholders represented by a question mark (?). The question mark placeholders are replaced with the actual values when the prepared statement is executed by the database. This approach makes it easier to read the prepared statement and identify any swaps or multiple assignments that are taking place. The second output allows for a clear visualization of the actual values that were inserted in place of the placeholders, thereby providing valuable insights into the behavior of the prepared statement.

We'll cover other logging possibilities later when discussing O/R mapper in Chapter 6, Section 6.3.2, so that we can do without P6 in principle. However, the solution is very neat and offers a great wealth of configurations. Only the formatting of the SQL statements isn't built in, but Hibernate can take care of that.

5.3.6 JDBC org.springframework.jdbc Package and Its Subpackages

The JDBC package `org.springframework.jdbc`[4] of the Spring Framework is one of the oldest packages. Rod Johnson already knew back then that the JDBC API is heavy and wanted to simplify database access. For this purpose, he introduced several subpackages, most importantly:

- **`org.springframework.jdbc.core`**
  This important subpackage contains types such as `JdbcTemplate`.
- **`org.springframework.jdbc.datasource`**
  This subpackage is primarily about building `DataSource` objects.
- **`org.springframework.jdbc.object`**
  This subpackage contains classes, for example, to make comfortable SQL inserts without much programming. The package name `object` indicates that this is more object-oriented than, for example, the `JdbcTemplate`.
- **`org.springframework.jdbc.support`**
  This subpackage contains utility classes and interfaces, for example, to manage automatically generated keys or translators for exceptions.

The root package `org.springframework.jdbc` contains only exception classes.

5.3.7 DataAccessException

When a regular Java program uses JDBC to interact with databases, it may encounter errors that result in a `java.sql.SQLException` being thrown. It's important to note that `SQLException` is a checked exception, which means that it must be declared in the method signature or handled by the calling method.

However, Spring takes a more abstract approach and questions the use of `SQLException` as the name of the exception. This is because NoSQL databases may also report similar exceptions, which are comparable to `SQLException`. Therefore, using `SQLException` as the name of the exception may not be appropriate for NoSQL databases, which don't use SQL. Recognizing the need for a more generic and inclusive exception naming convention that can be used across different types of databases and technologies, the Spring Framework introduced a new exception type: `DataAccessException`[5] (see Figure 5.13). The type expresses that something went wrong during data access. This exception type includes various subclasses that aren't just applicable to relational databases. This approach allows for a more generalized and inclusive exception handling mechanism that can be used across different types of databases and technologies.

---

[4] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/*

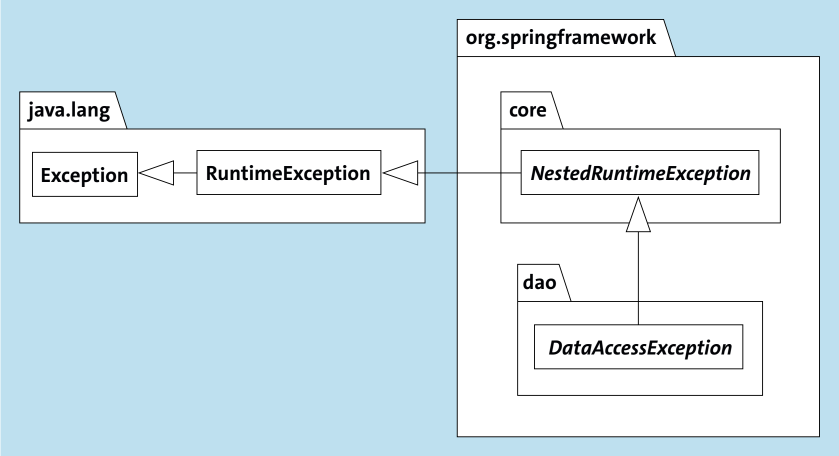


Figure 5.13 Superclasses of "DataAccessException"

`DataAccessException` is a `RuntimeException` as usual because Spring uses checked exceptions in almost no places. `DataException` is extended by a large variety of classes. There are more subclasses in the packages:

- **`org.springframework.dao`**
  These exceptions are general in nature, having to do with the implementation of data access objects (DAOs), that is, the implementation of a data access layer, but not specific to SQL.[6]
- **`org.springframework.jdbc`**
  With relational databases and SQL queries, a lot can go wrong, and there are a lot of exceptions. `BadSqlGrammarException` is a much more expressive name.[7]

---

[5] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/dao/DataAccessException.html*

[6] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/dao/package-tree.html*

[7] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/jdbc/package-tree.html*

org.springframework.jdbc.support.SQLExceptionTranslator

The JDBC API essentially expresses everything via the exception of type `SQLException` except for a tiny number of subclasses such as `BatchUpdateException` and `SQLRecoverableException`. With the JDBC API, a program must request the error code and identify from what the problem was from that error code.

Spring takes a much more object-oriented approach and uses the possibility that exception objects can be in an inheritance hierarchy. Spring translates these status codes of the respective databases into the corresponding exception. This is a challenge because the error codes are highly database-dependent.

The translation of the SQL error codes into the exceptions is done by an implementation of `org.springframework.jdbc.support.SQLExceptionTranslator`.[8] By default, Spring reads in an XML file for this purpose, which can be viewed at:
https://github.com/spring-projects/spring-framework/blob/main/spring-jdbc/src/main/resources/org/springframework/jdbc/support/sql-error-codes.xml

Here's an excerpt:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN"
  "https://www.springframework.org/dtd/spring-beans-2.0.dtd">
<!--…-->
<beans>
 <bean id="DB2" name="Db2" class=
"org.springframework.jdbc.support.SQLErrorCodes">
  <property name="databaseProductName">
   <value>DB2*</value>
  </property>
  <property name="badSqlGrammarCodes">
   <value>-007,-029,-097,-104,-109,-115,-128, … , -491</value>
  </property>
  <property name="duplicateKeyCodes">
   <value>-803</value>
  </property>
  …
 </bean>
 …
 <bean id="H2" class="org.springframework.jdbc.support.SQLErrorCodes">
  <property name="badSqlGrammarCodes">
   <value>42000,42001,42101,42102,42111,42112,42121,42122,42132</value>
  </property>
  <property name="duplicateKeyCodes">
```

---

[8] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/jdbc/support/SQLExceptionTranslator.html*

```
   <value>23001,23505</value>
  </property>
  …
 </bean>

</beans>
```

5.3.8 Auto-Configuration for DataSource *

Spring has quite a few auto-configurations. One particular `DataSourceAutoConfiguration`[9] leads to a Spring-managed bean `DataSource` if we haven't built a `DataSource` ourselves. The configuration parameters from `spring.datasource.*` are mapped to a special property object, and a look at the source code of the `DataSourceAutoConfiguration` class[10] shows the typical conditions we already learned about in Chapter 2, Section 2.10.

Build DataSource Yourself

The auto-configuration is handy, but you can build your own `DataSource` object and put it in context as a Spring-managed bean instead. The construction is easily done using the `DataSourceBuilder`.[11] Here's an example:

```
@Configuration
class DataSourceConfig {
  @Bean
  public DataSource getDataSource() {
    return DataSourceBuilder.create()
        // .driverClassName( "org.h2.Driver" )
        .url( "jdbc:h2:tcp://localhost/~/unicorns" )
        .username( "user" )
        .password( "pass" )
        .build();
  }
}
```

The static method `create()` on the `DataSourceBuilder` returns a `DataSourceBuilder`, and connection parameters such as JDBC URL, username, password, or class name can be set for the driver—the latter isn't necessary in most cases. At the end, the `build()`

---

[9] *https://docs.spring.io/spring-boot/docs/current/api/org/springframework/boot/autoconfigure/jdbc/DataSourceAutoConfiguration.html*

[10] *https://github.com/spring-projects/spring-boot/blob/main/spring-boot-project/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/jdbc/DataSourceAutoConfiguration.java*

[11] *https://docs.spring.io/spring-boot/docs/current/api/org/springframework/boot/jdbc/DataSourceBuilder.html*

method returns the `DataSource` that an `@Bean` method will set in the context. Then, the rule is that once we've built the `DataSource`, the auto-configuration doesn't do it anymore.

5.3.9 Addressing Multiple Databases

By default, Spring Boot maps configuration data to a `DataSourceProperties` object via a `@ConfigurationProperties(prefix="spring.datasource")` object.[12] However, Spring Boot supports only a single preconfigured data source. If a program wants to use multiple `DataSources`, a small detour is required. There are different ways to achieve this, and we'll show you two of them in the following subsections.

More Than a Database: Programmed

The first way is to also build multiple `DataSource` instances via multiple `@Bean` methods and name them:

```
@Configuration
class DataSourceConfig {

  @Bean
  @Primary
  public DataSource mainDataSource() {
    return …
  }

  @Bean( "datasource2" )
  public DataSource secondaryDataSource() {
    return …
  }
}
```

The main database source is annotated with `@Primary` so that it doesn't have to be named—you learned about this annotation in Chapter 2, Section 2.7.3. A `@Primary` bean has the advantage that it's automatically wired whenever a `DataSource` bean is needed, even if there are multiple `DataSource`-type Spring-managed beans in the context. The second `DataSource` has an explicit name, but we know it always has a name by the method name. Of course, other qualifiers besides the name are possible, and an annotation works just as well.

---

[12] *https://docs.spring.io/spring-boot/docs/current/api/org/springframework/boot/autoconfigure/jdbc/DataSourceProperties.html*

If named `DataSource` objects are in the context as Spring-managed beans, a bean with the name can be requested in the next step during wiring:

```
@Autowired
@Qualifier( "datasource2" )
DataSource dataSource;
```

If the `DataSource` itself isn't relevant at all, but only other objects are configured with it, for example, the `JdbcTemplate`, then a constructor injection is nice because it also works with a `@Qualifier`. An example with the `JdbcTemplate` looks like this:

```
MyRepository( @Qualifier( "datasource2" ) DataSource dataSource ) {
  jdbcTemplate = new JdbcTemplate( dataSource );
}
```

The `@Bean` methods built the `DataSource` objects, and, in principle, you could fall back on external configurations—but this is easier to do.

More Than One Database: Configured Externally

If we want to configure a database externally, we first have to choose a prefix behind which the configuration values are placed. For example, let's take `spring.datasource2`:

```
spring.datasource2.url=…
spring.datasource2.username=…
spring.datasource2.password=…
```

Whether it's clever to pollute the namespace of Spring, however, is another question.

The second step is to write an `@Bean` factory method that, via `DataSourceBuilder.create().build()`,[13] returns an empty `DataSource`:

```
@Bean( "datasource2" )
@ConfigurationProperties( prefix = "spring.datasource2" )
public DataSource dataSource2() {
  return DataSourceBuilder.create().build();
}
```

Because the `@Bean` method is annotated with `@ConfigurationProperties`, Spring Boot calls setters such as `setUrl(…)` and `setPassword(…)` on the `DataSource`. It's not so common that `@Bean` methods are annotated with `ConfigurationProperties`, but here it's convenient. We discussed the procedure in Chapter 3, Section 3.2.6.

---

[13] *https://docs.spring.io/spring-boot/docs/current/api/org/springframework/boot/jdbc/DataSourceBuilder.html*

Note

While the `DataSourceProperties` contain the properties for `spring.datasource.*` but aren’t a `DataSource` instances themselves, `@ConfigurationProperties` really transfer the data to the `DataSource`.

5.3.10 DataSourceUtils *

With a `DataSoure` in hand, a JDBC connection can be established with `getConnection()`, but this introduces two problems. When using the JDBC API directly, there are still checked exceptions, and a `Connection` might already be open with a transaction that you want to continue. Two methods of class `DataSourceUtils`[14] help:

- **`Connection doGetConnection(DataSource dataSource) throws SQLException`**
  This gets the JDBC connection of the `DataSource` and returns it. Exceptions are thrown as checked `SQLException`.
- **`Connection getConnection(DataSource dataSource) throws CannotGetJdbcConnectionException`**
  This calls `doGetConnection(…)` internally, but translates an `SQLException` to an unchecked `CannotGetJdbcConnectionException`.

The error handling during construction is provided by `getConnection(…)`. For the mapping of concrete SQL exceptions, a `SQLExceptionTranslator` can be used:

```
SQLExceptionTranslator sqlExceptionTranslator =
    new SQLExceptionSubclassTranslator();

var connection = DataSourceUtils.getConnection( dataSource );
try {
  var statement = connection.createStatement();
  …
}
catch ( SQLException e ) {
  throw sqlExceptionTranslator.translate(
     "readable text describing the task being attempted",
     "the SQL query or update that caused the problem (if known)", e );
}
finally {
  DataSourceUtils.releaseConnection( connection, dataSource );
}
```

---

[14] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/jdbc/datasource/DataSourceUtils.html*

The `getConnection(…)` method is responsible for catching `SQLException`s and translating them into Spring's `DataAccessException` subclasses. The `try-catch` block is used to encapsulate our code that uses the JDBC API, from the `Connection` to the `[Prepared]Statement`, and then to the `ResultSet`. The `catch` block catches any checked exceptions that occur and translates them into Spring's exceptions using the `sqlExceptionTranslator`. The `translate(…)` method is passed three arguments: a description of the problem, the SQL string, and the `SQLException` object. In this case, the description is the only argument passed.

After the JDBC API has been used, the connection must be closed. The `close()` method of the connection can also throw an `SQLException`, so the program must translate it again. To avoid this, the program uses the `releaseConnection(Connection con, DataSource dataSource)` method of `DataSourceUtils`, which internally calls a `doReleaseConnection(…)` method that throws `SQLException`. If an error occurs, the `releaseConnection(…)` method catches and logs any exceptions. If the connection isn't currently in an active transaction, it will be released. This method isn't named `close()` because if the connection is still used in a transaction, it can't be closed yet. Transactions will be discussed in more detail in Section 5.11.

5.4 JdbcTemplate

We've briefly covered the fundamentals of accessing data using the `JdbcTemplate`. Using `JdbcTemplate` is significantly more convenient than using the pure JDBC API. Overall, `JdbcTemplate` has several advantages:

- Conveniently executes SQL statements
- Reads results and transforms them into objects
- Converts and wraps `SQLException` in `DataAccessException`
- Thread-safe once configured with a data source
- Logs all SQL errors with the `DEBUG` level by default

The class `JdbcTemplate` is very extensive with almost 100 methods (some other methods have been deprecated as well). We'll take a look at the central methods of the class in the following sections.

5.4.1 Execute Any SQL: execute(…)

The most general method for executing arbitrary SQL statements is `execute(String)`. The method is useful when arbitrary scripts are to be executed. For example, if an initialization script is to be processed, it can be read in and passed to the `execute(…)` method.

As a second example, the `Profiles` table is to be deleted from the database:

```
jdbcTemplate.execute( "DROP TABLE Profile" );
```

In addition, for *Data Definition Language* (DDL) operations, `execute(…)` is well suited.

For individual queries, insertion operations, or deletion operations, you'll resort to more specialized methods.

5.4.2 SQL Updates: update(…)

The `update(…)` methods are useful for updating and inserting data. There are also a number of variants for batch processing. More about this will follow later in Section 5.7.

Like many other methods of the `JdbcTemplate` class, `update(…)` is overloaded. The basic structure is as follows:

```
int update(String sql [,…])
```

All `update(…)` methods share a common feature, which is returning the number of affected rows. If the JDBC driver is unable to receive this information from the database, a magic value of `-2` is returned. A practical example of this method is deleting all unicorns from our sample database that haven't logged in for a year and finding out how many of them were deleted.

5.4.3 Query Individual Values: queryForObject(…)

For SQL queries, `JdbcTemplate` declares about 40 `query*(…)` methods. A simple method `queryForObject(…)` can be used if the result consists of exactly one single element:

```
<T> T queryForObject(String sql, Class<T> requiredType [,…])
```

Again, first is the SQL string, and the `Class` object is for the type token, which also determines the return type. For example, the number of rows is returned by a `SELECT COUNT(*)`, and the end is an integer:

```
int size = jdbcTemplate.queryForObject( "SELECT COUNT(*) FROM Profile",
                                         Integer.class );
```

The integer is designated as `Integer.class`, but Unboxing converts the `Integer` object to a primitive `int`.

The query for columns is a good example of `queryForObject(…)` because the return must consist of only one row and one column.

If the SQL statement returns no row or more than one row, the method throws an `IncorrectResultSizeDataAccessException`. In addition, there can be a `DataAccessException` if something was wrong with the query.

The `queryForObject(…)` method has a `requiredType`, and the parameter appears at many methods:

```
<T> T queryForObject(String sql, Class<T> requiredType [,…])
```

As a generically typed `Class` object, the `requiredType` gives the methods a return type. We use it to additionally express which Java type is to be mapped to the database type. If the mapping isn't possible because of incompatible types, an `InvalidDataAccessApiUsageException` is thrown.

5.4.4 Define a Placeholder for a PreparedStatement

Suppose we have a GUI that allows users to input a name, which is stored in a variable called `name`. If we want to search for this name in the database, we can use the following approach:

```
var name = …;         // Input from the GUI
var sql  = "SELECT * FROM Profile WHERE nickname LIKE '" + name + "'";
```

The name is built into the query with a plus sign, but is that good?

This solution is terrible, for a whole host of reasons:

1. Strings concatenated with plus are difficult to read and maintain. Programming errors arise quickly because spaces or apostrophes are missing.
2. If the name itself contains apostrophes, the simply concatenated string at the end is no longer a valid SQL statement. For example, when entering `O'Neil's` in the concatenation, `… LIKE '" + "O'Neil's" + "'"` results in `… LIKE 'O'Neil's'`, and that is broken SQL.
3. Always building new SQL statements puts pressure on the database because each new SQL statement is always parsed, optimized, and later cached.
4. The greatest danger comes from *SQL injections* emanating. These are cleverly formulated SQL queries with malicious SQL statements, such as the following:

   ```
   var name = "'; DROP ALL OBJECTS; SELECT '"
   var sql  = "SELECT * FROM Profile WHERE nickname LIKE '" + name + "'";
   ```

If attackers actually manage to execute such SQL strings, the database is quickly deleted. Many attacks can be traced back to SQL injections. The *Open Web Application Security Project* (OWASP) lists SQL injections in its top 10 attacks.[15]

A `PreparedStatement` offers a solution for all listed problems, that is, a prepared statement. As a reminder, the RDBMS will compile and provide the `PreparedStatement` once, and only the data will be added later. Strings concatenated with plus, on the other hand, are always new, and this would not be optimal for performance reasons.

This

```
"SELECT * FROM profiles WHERE nickname LIKE '" + name + "'"
```

---

[15] See *https://owasp.org/www-project-top-ten*.

becomes

```
"SELECT * FROM Profile WHERE nickname LIKE ?"
```

Almost every method from the `JdbcTemplate` can be passed an SQL string with the placeholder `?`. The question mark is a placeholder for data that will be passed in a second step when sending.

Here's an example: the mane length of a profile of a unicorn with a freely selectable name is to be determined:

```
var sql = "SELECT manelength FROM Profile WHERE nickname = ?";
int len = jdbcTemplate.queryForObject( sql, Integer.class, "FillmoreFat" );
```

In some cases, the arguments aren't passed as variable arguments (varargs), but as `Object[] args`.

5.4.5 Query Whole Row: queryForMap(…)

The method `queryForObject(…)` returns exactly one result from exactly one row with one column. To read data from a row with multiple columns, `JdbcTemplate` provides the following overloaded methods:

- `Map<String, Object> queryForMap(String sql)`
- `Map<String, Object> queryForMap(String sql, Object... args)`
- `Map<String, Object> queryForMap(String sql, Object[] args, int[] argTypes)`

The parameterization reveals that the SQL string is passed first. In the second method, varargs can again be used for the prepared statements, and in the third variant, the `PreparedStatement` types are explicitly defined with values from `java.sql.Types`.

The `queryForMap(…)` method returns an associative data structure in which column names are associated with the respective assignments of the column. Because the column names are always strings, the key type is `String`. The contents of the columns can be arbitrary, so the return type is `Map<String, Object>`.

Consider this brief example where we want to query the `nickname` and `lastseen` columns from a profile with a specified ID:

```
var sql = "SELECT nickname, lastseen from Profile WHERE id = ?";
Map<String, Object> result = jdbcTemplate.queryForMap( sql, 1 );
```

The result is an associative data structure that can be queried using the usual methods: `result.get("nickname")` and `result.get("lastseen")`.

As with `queryForObject(…)`, the result may only consist of exactly one row with `queryForMap(…)`; however, with `queryForMap(…)` there could be more columns than with `queryForObject(…)`.

5.4.6 Query Multiple Rows with One Element: queryForList(…)

If queries return multiple rows with one column, the results can be retrieved with `queryForList(…)`:

- `<T> List<T> queryForList(String sql, Class<T> elementType)`
- `<T> List<T> queryForList(String sql, Class<T> elementType, Object... args)`

The parameter types are similar to `queryForMap(…)`, except that the return is a list.

Here's an example: When are the different birthdays of the 10 oldest unicorns?

```
var sql = """
          SELECT DISTINCT birthdate
          FROM            Profile
          ORDER BY        birthday
          LIMIT           ?""";

// List<Timestamp> dates=jdbcTemplate.queryForList(sql, Timestamp.class, 10 );
List<LocalDate> dates = jdbcTemplate.queryForList(sql, LocalDate.class, 10 );
```

The SQL statement takes the `Profiles` table, sorts by `birthday`, and removes duplicate birthdays with `DISTINCT`. At the end, there is a list of date values. The number `10` is parameterized. For the date-time values, `LocalDate` and the "old" data type `java.sql.Timestamp` (which is JDBC compliant) can be specified as type tokens. Spring allows the Java Date-Time API, so we'll use it. The return type is then `List<LocalDate>`.

5.4.7 Read Multiple Rows and Columns: queryForList(…)

The `JdbcTemplate` has other variants of `queryForList(…)` where the return consists of multiple rows with multiple columns, so it's conceptually a table:

- `List<Map<String,Object>> queryForList(String sql)`
- `List<Map<String,Object>> queryForList(String sql, Object... args)`

The meaning of the type `List<Map<…>>` is that it represents a list containing multiple mini-maps, where each mini-map maps a column name to its corresponding value in the column.

Here's an example: A date is given as a string in the form `YYYY-MM-DD` as input in variable `date`. A program should display the nickname and time of all unicorns that have logged in after the given date:

```
var date = "1980-01-01";
var sql = """
          SELECT   nickname, lastseen
          FROM     Profile
          WHERE    lastseen > ?
```

```
         ORDER BY lastseen""";

jdbcTemplate.queryForList( sql, LocalDate.parse( date ) )
    .stream()
    .map( columnsValueMap -> {
      record NicknameLastseen(String name, LocalDateTime seen) { }
      return new NicknameLastseen(
          "" + columnsValueMap.get( "nickname" ),
          ((Timestamp)columnsValueMap.get("lastseen")).toLocalDateTime() );
    } )
    .map( data -> data.name + ", " + data.seen )
    .forEach( System.out::println );
```

The `queryForList(…)` method is given the SQL, and the second parameter is for the placeholder; to do this, the program converts the `String` into a `LocalDate` object. (Strictly speaking, there is no need to convert the string at all because internally the string is automatically converted to a `Timestamp`). We can pass `LocalDate` objects, but, unfortunately, we get back `Timestamp` objects.

The result of the `queryForList(…)` method is the list of mini-maps. To transform the data, a stream is built, and a `Stream<Map<…>>` is created. The `map(…)` method transforms the entries (i.e., the rows from the SQL query) into `NicknameLastseen` objects.[16] The passed `Function` extracts the columns `nickname` and `lastseen` from the `Map` and calls the record constructor with them. The date value of `java.sql.Timestamp` is converted into a `LocalDateTime`.

The `Stream<NicknameLastseen>` can be evaluated. For easy output, the program maps the record components to a string which is output. Running the elements and transferring the columns to objects sounds like something that can be automated—and that's precisely what the `JdbcTemplate` can do.

5.5 Data Types for Mapping to Results

Up to this point, we've focused on the methods of the `JdbcTemplate` class that use JDBC data types. However, as software developers operating in an object-oriented environment, we prefer to work with business objects or typed containers rather than generic maps or lists of maps. In the first step, we created a map ourselves, but relying on external iteration (loops) suggests that the framework should handle this functionality. Therefore, we'll explore various data types that are designed primarily to convert a JDBC `ResultSet` into an object (see Figure 5.14).

---

[16] It's little known that you can declare and return local data types such as a record, but here it's a useful feature.

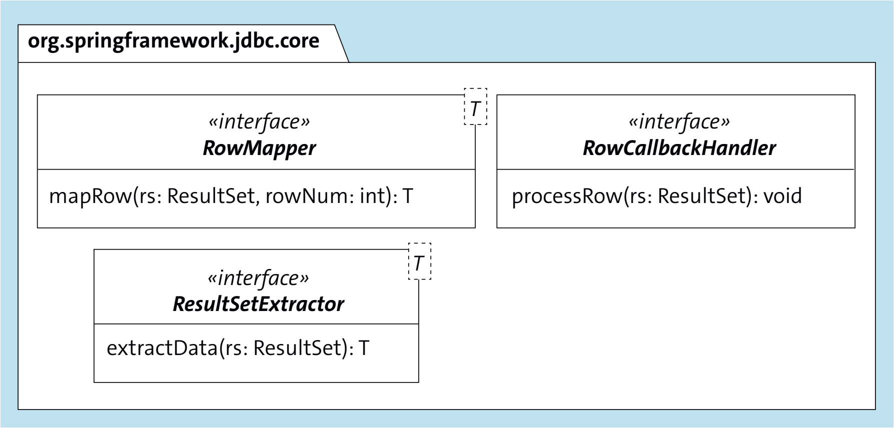

org.springframework.jdbc.core
«interface»
RowMapper
T
mapRow(rs: ResultSet, rowNum: int): T
«interface»
RowCallbackHandler
processRow(rs: ResultSet): void
«interface»
ResultSetExtractor
T
extractData(rs: ResultSet): T

Figure 5.14 Three Selected Interfaces for Internal Iteration via a "ResultSet"

The role of the `RowMapper` is to convert a row from a JDBC result set into an object. In contrast, the `RowCallbackHandler` doesn't return anything but rather "consumes" the row itself. For instance, it may be used to write the `ResultSet` to a file. The `ResultSetExtractor` allows us to manually iterate over the entire `ResultSet` to generate a completely separate object at the end. This approach is helpful when there isn't a one-to-one mapping between a row and a result, but rather the entire `ResultSet` needs to be processed and mapped to a single object as the result.

5.5.1 RowMapper

A `RowMapper`[17] is a functional interface with a method `mapRow(…)`:

```
@FunctionalInterface
public interface RowMapper<T> {
    @Nullable
    T mapRow(ResultSet rs, int rowNum) throws SQLException;
}
```

In essence, it's similar to a `BiFunction`:[18] two inputs are provided, and one output is returned. Spring takes care of running the `ResultSet` for us and provides our `RowMapper` implementation with two inputs: the current row and its corresponding row number. The role of the `RowMapper` implementation is to extract the data from the columns of the row in the `ResultSet` and convert it into an object.

---

[17] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/jdbc/core/RowMapper.html*

[18] Because `SQLExceptions` can occur, it can't really be a `BiFunction`.

Pass RowMapper

There are different places where a `RowMapper` can be passed, primarily in `JdbcTemplate`, but also in `NamedParameterJdbcTemplate` and even in *Spring Data JDBC*.[19] Here is an excerpt of the methods from `JdbcTemplate`:

- `<T> T queryForObject(String sql, RowMapper<T> rowMapper)`
- `<T> T queryForObject(String sql, RowMapper<T> rowMapper, Object... args)`
- `<T> List<T> query(String sql, RowMapper<T> rowMapper)`
- `<T> List<T> query(String sql, RowMapper<T> rowMapper, Object... args)`

The return value of these methods is determined by the output of the `RowMapper` that is passed as an argument. The `queryforObject(…)` method only returns a single object, meaning that a single row is converted into an object using the `RowMapper`. Other calling conventions are the same, and placeholders can also be used for a prepared statement.

The `query(…)` method has the most concise name for querying, and it returns a list of elements based on the `RowMapper` provided as an argument. This approach is especially useful when the query result can be empty.

Example with RowMapper

We need to implement the `RowMapper` functional interface. Spring iterates over the `ResultSet` for us from top to bottom and calls the method `mapRow(…)` on the `RowMapper`. This is a perfect division of labor: there is the internal iteration in the Spring Framework on one side, and our job is just to make an object out of the `ResultSet`.

Here's an example scenario where we wanted to know which profiles with which names were last seen and when. The corresponding SQL query is as follows:

```
var sql = """
          SELECT   nickname, lastseen
          FROM     Profile
          WHERE    lastseen > ?
          ORDER BY lastseen""";
```

The record `NicknameLastseen` is to be used again as the result container:

```
record NicknameLastseen(String name, LocalDateTime seen) { }
```

With the `JdbcTemplate` method `query(…)` and a `RowMapper`, the code becomes much shorter:

```
var date = "1980-01-01";
List<NicknameLastseen> profiles = jdbcTemplate.query(
    sql,
    (rs, __) -> new NicknameLastseen(
```

---

[19] *https://docs.spring.io/spring-data/jdbc/docs/current/reference/html/#jdbc.query-methods.at-query.custom-rowmapper*

```
                rs.getString( "nickname" ),
                rs.getTimestamp( "lastseen" ).toLocalDateTime() ),
    date );
```

The method call requires three arguments: the SQL statement, the `RowMapper` (implemented using a lambda expression), and the prepared statement assignment (`date`).

The implementation is highly concise due to the lambda expression. The row number is unnecessary in this scenario, so it's represented with two underscores. Inside the `RowMapper`, the `NicknameLastseen` object is constructed, and the constructor is supplied with the column contents. Because JDBC provides a `java.sql.Timestamp`, but the record requires a `LocalDateTime`, a conversion from the JDBC column type is necessary.

The framework handles the task of collecting the objects in a list, which greatly reduces the amount of code needed. Additionally, a `RowMapper` is typically stateless and can be reused elsewhere. For example, a `RowMapper` can be used with `queryForObject(…)` when only a single object is required instead of a list.

BeanPropertyRowMapper Implementations

Of the `RowMapper` interface, Spring provides prebuilt implementations (see Figure 5.15).

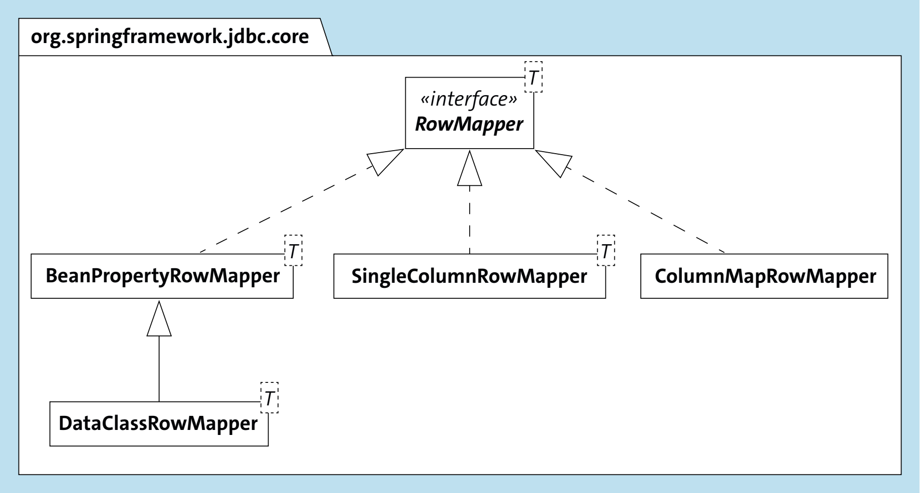


Figure 5.15 Selected "RowMapper" Implementations

To gain a better understanding, let's examine two row mapper types in detail: `BeanPropertyRowMapper` and its subtype, `DataClassRowMapper`.

BeanPropertyRowMapper: An Example

The class name `BeanPropertyRowMapper`[20] already suggests what it's about: JavaBeans. As a reminder, JavaBeans are classes that have a parameterless constructor as well as setters/getters for properties. If an application uses JavaBeans, the `BeanPropertyRowMapper` can transfer the columns directly to the properties of these beans. For example, given a JavaBean for a subset of photo information:

---

[20] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/jdbc/core/BeanPropertyRowMapper.html*

```
public class ProfilePhoto {
  private long   id;
  private String imageName;
  public long getId() { return id; }
  public void setId( long id ) { this.id = id; }
  public String getImageName() { return imageName; }
  public void setImageName( String imageName ) { this.imageName=imageName; }
}
```

The `query(…)` method can use a `RowMapper` (and the `BeanPropertyRowMapper` is a special `RowMapper`) to return a `List` of `ProfilePhoto` instances:

```
List<ProfilePhoto> photos = jdbcTemplate.query(
  "SELECT id, name AS imageName FROM Photo WHERE is_profile_photo=TRUE",
  new BeanPropertyRowMapper<>( ProfilePhoto.class )
);
```

The `BeanPropertyRowMapper` transfers columns directly to an object if the properties are named the same as the columns. In the given `SELECT` statement, the `id` and `name` columns are selected, but an important adjustment is needed because the column `name` doesn't match our property `imageName`. Thus, the SQL statement uses an `AS` keyword to create a column alias that matches the property name for `BeanPropertyRowMapper`.

When `BeanPropertyRowMapper` is passed a type token for the JavaBean, it calls the parameterless constructor of the `ProfilePhoto` class, extracts all columns, retrieves their values, and transfers them to the new instance using the setters. This is why a JavaBean is necessary, as there needs to be some kind of interface that the `BeanPropertyRowMapper` can automatically call. Setters are therefore compulsory, while getters aren't strictly necessary because they aren't called by Spring, although it's reasonable to offer getters for symmetry reasons.

DataClassRowMapper

JavaBeans aren't as significant today as they were in the past, as records have become more popular. To accommodate this change, the Spring team has introduced a subclass of the `BeanPropertyRowMapper` class called `DataClassRowMapper`.[21] This mapper works with both JavaBeans and records, which don't require setters or getters.

---

[21] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/jdbc/core/DataClassRowMapper.html*

For example, the goal is to map only `nickname` and `lastseen` from the `Profiles` table to a record `NicknameLastseen`:

```
record NicknameLastseen(String name, LocalDateTime seen) { }
```

The record components are named `name` and `seen`, not `nickname` and `lastseen`, so the SQL statement must rename the columns with `AS` again:

```
var date = "1980-01-01";
var sql = """
          SELECT   nickname AS name, lastseen AS seen
          FROM     Profile
          WHERE    lastseen > ?
          ORDER BY lastseen""";

List<NicknameLastseen> list = jdbcTemplate.query(
    sql, new DataClassRowMapper<>( NicknameLastseen.class ), date );
```

The calling convention is the same as before: a type token is passed to the constructor of `DataClassRowMapper`, along with the date and parameter assignments from the prepared statement. This results in a very concise query, with minimal source code required for mapping rows to Java objects. Spring handles the iteration as well, so ultimately all that is needed is SQL and a suitable container. However, refactoring can still present a challenge, as renaming a column or record component may only be noticeable during testing or in production use.

queryForStream(…) for a Stream

In addition to the `queryForList(…)` and `queryForMap(…)` methods, the `JdbcTemplate` also has the `queryForStream(…)` methods, which return a `java.util.stream.Stream`:

- `Stream<T> queryForStream(String sql, RowMapper<T> rowMapper)`
- `Stream<T> queryForStream(String sql, RowMapper<T> rowMapper, Object... args)`
- `Stream<T> queryForStream(String sql, PreparedStatementSetter pss, RowMapper<T> rowMapper)`

All these methods expect a `RowMapper`, which means that the `Stream` always contains self-constructed objects and not any maps.

Note

With a stream, there is a live connection to the database, and it's important to close the stream. The best way to do this is with a `try`-with-resources block:

```
try ( Stream<T> s = jdbcTemplate.queryForStream( sql, rowMapper ) ) { … }
```

5.5.2 RowCallbackHandler

The `*Mapper` types presented so far convert a row into an object, and thus a simple mini O/R mapper can be written. If the data from the columns is processed directly and doesn't have to be transferred to objects, there is another callback type: `RowCallbackHandler`.[22] The functional interface prescribes the following:

```
void processRow(ResultSet rs) throws SQLException
```

The method `processRow(…)` receives only the `ResultSet` and no row counter; furthermore, the method returns nothing.

`RowCallbackHandlers` are used in places where a mapping to objects isn't necessary, but where programs themselves process the results of a row. `RowCallbackHandlers` are transmitted in various places, for example, in `query(…)` methods:

- `void query(String sql, RowCallbackHandler rch)`
- `void query(String sql, RowCallbackHandler rch, Object... args)`

In addition, the `query(…)` methods return nothing, and the return type is `void` because the `RowCallbackHandler` "consumes" the data itself.

For example, let's assume 10 random nicknames are to be assembled from the database into a string. The string should start with "Meet", the individual segments of the string should be separated by commas, and the string should end with "and so many more!":

```
var sql = "SELECT nickname FROM Profile ORDER BY RAND() LIMIT ?";
var joiner = new StringJoiner( ", ", "Meet ", " and so many more!" );

jdbcTemplate.query( sql, rs -> {
  joiner.add( rs.getString( "nickname" ) );
}, 10 );
```

To concatenate strings, the code uses the `StringJoiner` class, which is initialized with a prefix and suffix. The `query(…)` method takes the SQL statement, the `RowCallbackHandler` as a lambda expression, and `10` for the count as the third argument. The lambda block is enclosed in curly braces for readability purposes.

The implementation of the `RowCallbackHandler` interface that we created extracts the nickname from each row and adds it to the `StringJoiner`, which collects the names. Once the `query(…)` method is called, the result can be retrieved as a string via the `toString()` method.

To summarize, Spring invokes the `RowCallbackHandler` method for each row, and the program can perform any necessary operations with the row. However, it should not continue the `ResultSet`.

---

[22] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/jdbc/core/RowCallbackHandler.html*

5.5.3 ResultSetExtractor

With a `RowMapper` and `RowCallBackHandler`, the `JbcTemplate` iterates over the `ResultSet` for us, and we can process the row. Spring JDBC further declares the type `ResultSetExtractor`,[23] which provides us with the entire `ResultSet`; we iterate over all the rows ourselves and map to exactly one final result. The method of the `RowCallBackHandler` functional interface follows:

```
T extractData(ResultSet rs) throws SQLException, DataAccessException
```

Consider extending the earlier example, where rows are gathered using a `StringJoiner`:

```
var sql = "SELECT nickname FROM Profile ORDER BY RAND() LIMIT ?";
var string = jdbcTemplate.query( sql, rs -> {
  var joiner = new StringJoiner( ", ",
                                 "Meet ", " and so many more!" );

  while ( rs.next() )
    joiner.add( rs.getString( "nickname" ) );

  return joiner.toString();
}, 10 );
```

The program's functionality remains the same—it randomly selects nicknames. However, the implementation now uses a `ResultSetExtractor` defined as a lambda expression. The lambda expression iterates through the `ResultSet` using a while loop, extracts the nickname column, reads the value, and appends it to a `StringJoiner`. Finally, the `query(…)` method caller receives the resulting string.

PreparedStatementCreator in the Background

Our first database query was

```
var sql = "SELECT manelength FROM Profile WHERE nickname = ?";
int len = jdbcTemplate.queryForObject(sql, Integer.class, "FillmoreFat");
```

The vararg argument signals to Spring to always use a prepared statement, regardless of whether the SQL statement has a question mark or not. Spring's `JdbcTemplate` always uses prepared statements when executing SQL queries.

`PreparedStatementCreator` is the component responsible for constructing these prepared statements in Spring, and there is a default implementation that Spring uses if none is provided. While this default implementation always creates new `PreparedStatement` objects for each request, in practice, this doesn't usually result in a performance penalty because most JDBC drivers cache SQL statements with their prepared statements. Therefore, if the same SQL statement is encountered again, the JDBC driver automatically provides the cached prepared statement. This also highlights the importance of avoiding SQL statement concatenation with input as it can interfere with the caching mechanism.

---

[23] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/jdbc/core/ResultSetExtractor.html*

PreparedStatementCreator Interface

It may be necessary to build your own `PreparedStatement` objects. In such a case, the `PreparedStatementCreator` interface is implemented, and the instance is passed to the `JdbcTemplate` methods, which then fall back on their own `PreparedStatementCreator`.

`PreparedStatementCreator`[24] is a functional interface with one method:

```
java.sql.PreparedStatement createPreparedStatement(Connection connection)
```

For our own program to build `PreparedStatement` objects, access to the underlying JDBC connection is necessary, which is why Spring passes exactly this `Connection` to the method. Our task is to return a `PreparedStatement`. A symbolic implementation looks like this:

```
PreparedStatementCreator creator = (Connection connection) -> {
  PreparedStatement stmt = connection.prepareStatement( … );
  …
  return stmt;
};
```

The Lambda expression takes the `Connection` object and calls the `preparedStatement(…)` method, but other variants are possible.

In everyday life, the `PreparedStatementCreator` type is rarely needed. The one place where the type is importantis when inserting new rows where the database provides an automatically generated key.

A `PreparedStatementCreator` can be passed in various methods. Here's a small selection:

- `query(PreparedStatementCreator psc, RowMapper<T> rowMapper)`
- `update(PreparedStatementCreator psc)`
- `update(PreparedStatementCreator psc, KeyHolder generatedKeyHolder)`

The `update(…)` method stands out with its `KeyHolder`. A keyholder is a container where Spring can put the automatically generated keys of newly inserted rows.

---

[24] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/jdbc/core/PreparedStatementCreator.html*

KeyHolder and GeneratedKeyHolder

The interface `KeyHolder`[25] is declared as follows:

```
package org.springframework.jdbc.support;
import …
public interface KeyHolder {

  @Nullable
  Number getKey() throws InvalidDataAccessApiUsageException;

  @Nullable
  <T> T getKeyAs(Class<T> keyType) throws InvalidDataAccessApiUsageException;

  @Nullable
  Map<String, Object> getKeys() throws InvalidDataAccessApiUsageException;

  List<Map<String, Object>> getKeyList();
}
```

The `getKeys()` method of the interface returns the automatically generated key, which is typically a numeric value. However, if the key isn't numeric, such as in the case of a server-generated Universally Unique Identifier (UUID), the `getKeyList(…)` method can be used to retrieve the value of the columns. This method returns a list containing maps that associate the columns with their respective values. A default implementation of the `GeneratedKeyHolder` interface is provided by Spring.

The `getKeys()` method returns the automatically generated key, which is generally a number. Because the key doesn't have to be numeric, there is another way to access the value of the columns: using the `getKeyList(…)` method. The list contains maps with an association between the column and the respective associated value. This is necessary if the key isn't a numeric value but, for example, a server-generated UUID. For the interface `KeyHolder`, Spring declares an implementation `GeneratedKeyHolder`.[26]

KeyHolder and GeneratedKeyHolder

Consider this example where we want to insert new rows into the `Photo` table. The corresponding SQL query might look like this:

```
var sql = """
    INSERT INTO Photo (profile_fk, name, is_profile_photo, created)
    VALUES (?, ?, ?, ?)""";
```

---

[25] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/jdbc/support/KeyHolder.html*

[26] *https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/jdbc/support/GeneratedKeyHolder.html*

All columns are occupied, but not the ID because the automatically generated key must not be specified as a column.

Let's add a new profile:

```
var profile_fk = 12;
var name = "unicorn001";
var isProfilePhoto = false;
var created = LocalDateTime.now();

PreparedStatementCreator preparedStmtCreator = connection -> {
  PreparedStatement stmt = connection.prepareStatement(
      sql, Statement.RETURN_GENERATED_KEYS );
  stmt.setInt( 1, profile_fk );
  stmt.setString( 2, name );
  stmt.setBoolean( 3, isProfilePhoto );
  stmt.setTimestamp( 4, Timestamp.valueOf( created ) );
  return stmt;
};

KeyHolder keyHolder = new GeneratedKeyHolder();
jdbcTemplate.update( preparedStmtCreator, keyHolder );
Number key = keyHolder.getKey();
// keyHolder.getKeyList().get(0).get("id");
```

To ensure that generated keys are returned, the program passes `Statement.RETURN_GENERATED_KEYS` as the second argument to `prepareStatement(…)`. However, often only `null` is returned.

The program then places the data at the appropriate positions in the constructed `PreparedStatement` and returns it via the `PreparedStatementCreator` method. This `PreparedStatement` can be passed to the `JdbcTemplate` `update(…)` method with a `KeyHolder` type container, where `KeyHolder` is an interface and `GeneratedKeyHolder` is a default implementation.

The `update(…)` method puts the key into the `KeyHolder`, which can be retrieved with `getKey()`. If the key is nonnumeric, `getKeyList(…)` can be used as an alternative. In the code, the program queries the first element of the list, retrieves a `Map`, and extracts the content of the `id` column, which is numeric in this case.

5.6 NamedParameterJdbcTemplate

With the `JdbcTemplate`, it's possible to pass parameters to prepared statements. However, using question marks as placeholders can make the SQL query less readable. For instance, when querying the length of a profile name with a given nickname, the SQL statement would look like this:

```
var sql = "SELECT manelength FROM Profile WHERE nickname = ?";
```

With one placeholder that's fine, but with multiple question marks, it's not very obvious. Thus, in addition to the `JdbcTemplate` class, there's another class named `NamedParameterJdbcTemplate`.[27] This class replaces the question mark with a descriptive name. The alternative implementation would appear as follows:

```
var sql = "SELECT manelength FROM Profile WHERE nickname = :name";
```

The auto-configuration also builds a `NamedParameterJdbcTemplate` alongside the `JdbcTemplate` and associates it with the current connection information. Therefore, an instance can be injected directly:

```
@Autowired
NamedParameterJdbcTemplate namedJdbcTemplate;
```

5.6.1 Type Relationships of *JdbcTemplate

The `JdbcTemplate` and `NamedParameterJdbcTemplate` classes have no type relationship between them. The only relationship is that the `NamedParameterJdbcTemplate` itself internally forwards to the `JdbcTemplate` (see Figure 5.16).

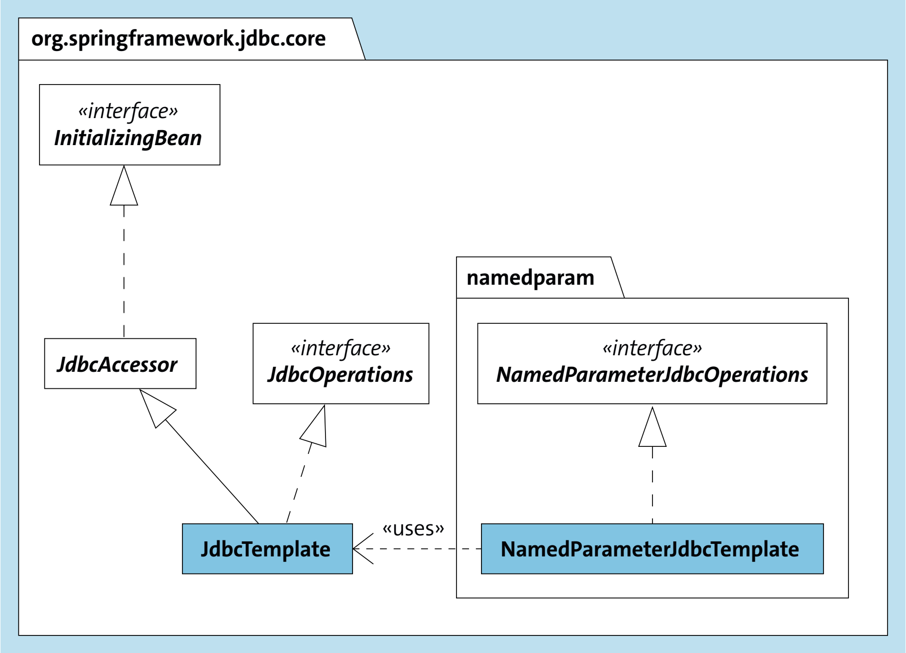

org.springframework.jdbc.core
«interface»
InitializingBean
namedparam
JdbcAccessor
«interface»
JdbcOperations
«interface»
NamedParameterJdbcOperations
JdbcTemplate
«uses»
NamedParameterJdbcTemplate

Figure 5.16 "JdbcTemplate" and "NamedParameterJdbcTemplate": No Common Base Types

---

[27] *https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/jdbc/core/namedparam/NamedParameterJdbcTemplate.html*

The core functionality of `NamedParameterJdbcTemplate` is to translate the *named* parameters into question marks and pass the query to the `JdbcTemplate`.

5.6.2 Methods of the NamedParameterJdbcTemplate

The methods between the two classes `JdbcTemplate` and `NamedParameterJdbcTemplate` are similar. There are the well-known `execute(…)` methods, `query*(…)` methods (e.g., `queryForStream(…)`, `queryForList(…)`, and `queryForMap(…)`), the `update(…)` methods, and `batchUpdate(…)` methods. In addition, the `ResultSetExtractor`, `RowCallbackHandler`, `RowMapper`, and `KeyHolder` types can be found.

One of the fundamental differences is the transmission of the parameters. With a `JdbcTemplate`, methods take the arguments with an array or vararg. That is, the first argument also occupies the question mark at position 1, the second argument occupies the question mark at position 2, and so on. With a `NamedParameterJdbcTemplate`, the order isn't significant. So, there must be a way to represent these parameters as key-value pairs.

There are two ways to pass parameters with the `NamedParameterJdbcTemplate`:

- The first option is to use a `Map<String, ?>`, where the key is the name of the parameter and the value is the assigned value.
- The second option is to use the `SqlParameterSource` object, which is also automatically used internally.

5.6.3 Pass Values of NamedParameterJdbcTemplate through a Map

Here is an example that demonstrates accessing the `NamedParameterJdbcTemplate` and passing the assignment of a placeholder `name` through a small `Map`:

```
var sql = "SELECT manelength FROM Profile WHERE nickname = :name";
int len = namedJdbcTemplate.queryForObject(
            sql, Map.of( "name", "FillmoreFat" ), Integer.class );
```

The first argument in `queryForObject(…)` is the SQL string as usual. Second, the method receives the container for the key-value pairs. The static `Map.of(…)` method helps to build a `Map` associating the parameter `name` with the assignment `"FillmoreFat"`. Because the mane length is an integer, the third argument, `Integer.class`, specifies the type information. The use of maps is the first variant, which is simple and syntactically compact.

5.6.4 SqlParameterSource

The second way to specify the key-value pairs is provided by the Spring data type `SqlParameterSource`.[28] The interface is implemented by several classes (see Figure 5.17).

---

[28] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/jdbc/core/namedparam/SqlParameterSource.html*

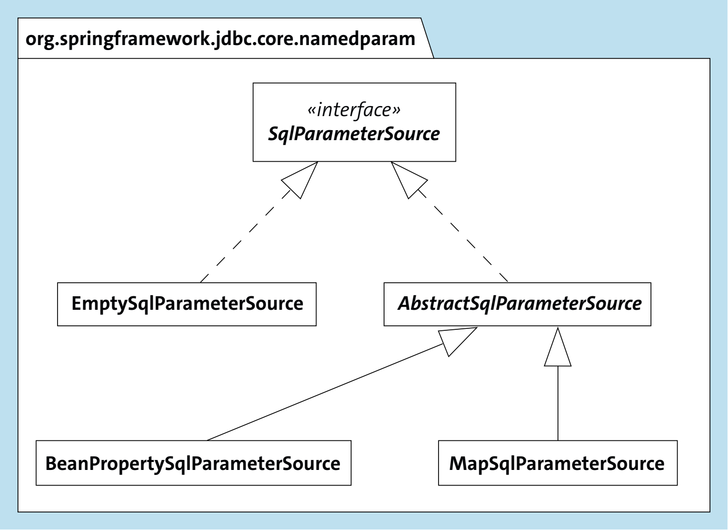

org.springframework.jdbc.core.namedparam
«interface»
SqlParameterSource
EmptySqlParameterSource
AbstractSqlParameterSource
BeanPropertySqlParameterSource
MapSqlParameterSource

Figure 5.17 Different Implementations of "SqlParameterSource"

One of them is `EmptySqlParameterSource`, which is used when no parameters are required. Another class, `MapSqlParameterSource`, serves as an adapter that allows a `Map` to be used as a `SqlParameterSource`. Additionally, `BeanPropertySqlParameterSource` uses reflection to provide mappings between a bean's properties and the corresponding `SqlParameterSource`.

An Example with MapSqlParameterSource

Consider the following two examples using `MapSqlParameterSource`:[29]

```
var sql = "SELECT manelength FROM Profile WHERE nickname = :name";
var map = new MapSqlParameterSource()
            .addValue( "name", "FillmoreFat" );
//          .addValue(…)
// var map = new MapSqlParameterSource( "name", "FillmoreFat" );
int len = namedJdbcTemplate.queryForObject( sql, map, Integer.class );
```

There are different ways to fill a `MapSqlParameterSource`:

- We could use the parameterless constructor and then add key-value pairs via the `addValue(…)` methods. This sets the key `name` and value.
- Key-value pairs can be passed in the constructor. This makes it well clear that `MapSqlParameterSource` is nothing more than an adapter of a `Map` to the `SqlParameterSource` from the Spring universe. Of course, further parameters can be added with `addValue(…)`.

---

[29] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/jdbc/core/namedparam/MapSqlParameterSource.html*

The second query provides an SQL update with two named parameters, `lastseen` and `name`:

```
var sql = """
          UPDATE Profile
          SET    lastseen = :lastseen
          WHERE  nickname = :name""";
var map = new MapSqlParameterSource( Map.of(
                                       "name",     "JimShu",
                                       "lastseen", LocalDateTime.now()
) );
namedJdbcTemplate.update( sql, map );
```

Of course, all key-value pairs must be set valid.

BeanPropertySqlParameterSource

The values of properties in a JavaBean can be retrieved using getters and set using setters. These properties can be used as a `SqlParameterSource` for a `NamedParameterJdbcTemplate`. The implementation of the `SqlParameterSource` interface, `BeanPropertySqlParameterSource`,[30] uses reflection to read the values via the getters.

Here's an example:

```
var sql = …;
SqlParameterSource params = new BeanPropertySqlParameterSource( sourceBean );
namedJdbcTemplate.update( sql, params );
```

The constructor of `BeanPropertySqlParameterSource` takes the filled JavaBean, and the `SqlParameterSource` object is passed to the usual methods `update(…)` and `query*(…)` as known.

The `BeanPropertySqlParameterSource` class performs the following steps internally:

1. It retrieves all the properties of the JavaBean that have corresponding setters and getters.
2. It calls the getters to obtain the values of the properties, and uses them to create key-value pairs that populate a new instance of `SqlParameterSource`.
3. With the filled `SqlParameterSource` object, the query can be executed.

Note: Performance Considerations

The `NamedParameterJdbcTemplate` internally builds a cache between the strings containing named parameters and the corresponding prepared statements, which makes recognizing the corresponding placeholders and building a regular prepared statement from them very performant.

---

[30] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/jdbc/core/namedparam/BeanPropertySqlParameterSource.html*

5.6.5 Access to Underlying Objects *

The `NamedParameterJdbcTemplate` internally accesses the `JdbcTemplate`. Because the `NamedParameterJdbcTemplate` can't do everything the `JdbcTemplate` can do, there are two methods that return the underlying `JdbcTemplate`:

- `JdbcTemplate getJdbcTemplate()`
- `JdbcOperations getJdbcOperations()`

`JdbcOperations` is the interface that implements `JdbcTemplate`, which is a bit more abstract.

If a client wants the flexibility to work with `NamedParameterJdbcTemplate` and `JdbcTemplate` at the same time, it's not necessary to inject two objects. One `NamedParameterJdbcTemplate` is sufficient. It's nicer with the readable parameter names anyway, and the client can query the `JdbcTemplate` if needed.

Getting from the `JdbcTemplate` to the `Connection` isn't intended, so there is no method that returns a `Connection`. In principle, however, the `JdbcTemplate` allows you to execute a piece of code in a `Connection` that has been set up. To accomplish this, you use the `execute(…)` method and pass a `ConnectionCallback`[31] object:

```
jdbcTemplate.execute( new ConnectionCallback<Object>() {
  @Override
  public Object doInConnection( Connection connection )
    throws SQLException, DataAccessException {
    …
    return …;
  }
} );
```

The `ConnectionCallback` functional interface declares a `doInConnection(…)` method. If we call `execute(…)` and pass our implementation, Spring will pass a `Connection` that our code block can use for database accesses.

The `doInConnection(…)` method can return results. Any SQL exceptions that occur within this block are caught and translated by Spring into the `DataAccessException` hierarchy. This exception translation is a major advantage over the approach of manually fetching a connection and working with it. This is because the JDBC API uses `SQLException`, and we would have to handle try-catch blocks ourselves.

The `execute(…)` method is overloaded. It exists in a second variant, not only with the `ConnectionCallback`, where a `Connection` is passed to the callback method, but a `StatementCallback`, which calls the callback method `doInStatement(Statement)`. This is a small shortcut. Occasionally, however, the `Connection` itself is necessary, for example, for metadata.

---

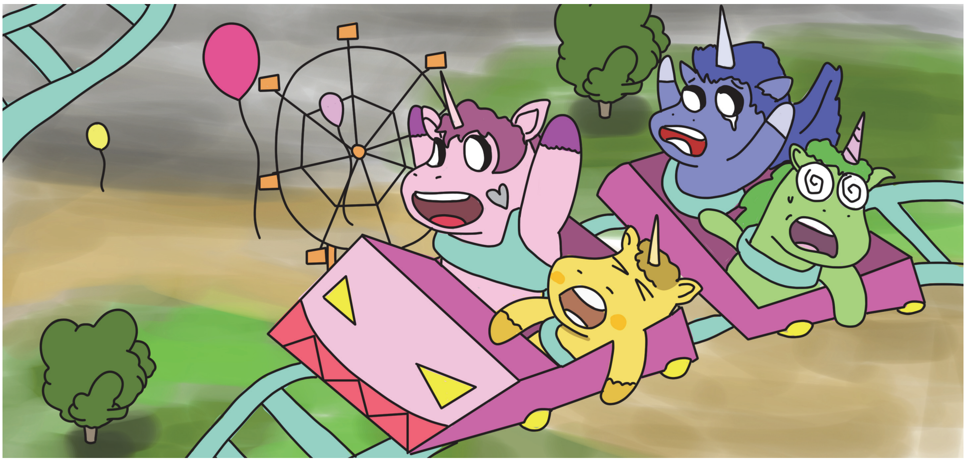
[31] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/jdbc/core/ConnectionCallback.html*

5.7 Batch Operations *

When a program needs to repeatedly perform small updates in a database, it can be inefficient and slow. This is analogous to going to the post office multiple times to collect packages instead of collecting all of them at once. In database operations, this is referred to as a *batch operation*. The use of batch operations is particularly relevant for batch inserts that involve inserting many new rows.

Consider a program that needs to store 1,000 rows in a database. Sending 1,000 individual prepared statements would not be optimal, even though it's better than rebuilding SQL `INSERT` statements without prepared statements. It would be faster to bundle these statements into *batches*. These batches can consist of, for example, 100 statements each, so that only 10 trips to the database are required. This approach saves bandwidth and optimizes memory usage.

The key advantage of using prepared statements and batch operations is that even large amounts of data can be inserted quickly. By taking advantage of these optimizations, programs can significantly improve their database performance.



5.7.1 Batch Methods for JdbcTemplate

The `JdbcTemplate` declares several `batchUpdate(…)` methods, such as the following:

- `int[] batchUpdate(String... sql)`
- `int[] batchUpdate(String sql, List<Object[]> batchArgs)`
- `int[] batchUpdate(String sql, List<Object[]> batchArgs, int[] argTypes)`

The easiest way to combine SQL statements into a bundle is provided by the `batchUpdate(String...)` method. The vararg contains SQL statements, as in the following example:

```
jdbcTemplate.batchUpdate(
  "INSERT INTO …",
  "INSERT INTO …"
);
```

The variant doesn't work with prepared statements; it only combines several SQL statements.

The two other `batchUpdate(…)` methods get a single SQL string and then an assignment of the prepared statement parameters. Instead of varargs, the data must be passed in a list.

The following is a sample call with separation of SQL and raw data:

```
List<Object[]> batchArgs = Arrays.asList(
  new Object[]{ … },
  new Object[]{ … }
);
jdbcTemplate.batchUpdate( "INSERT INTO … VALUES (?, …)",
                          batchArgs );
```

The return of the `batchUpdate(…)` methods is always an `int` array with information on how many rows were modified by each SQL statement.

5.7.2 BatchPreparedStatementSetter

The disadvantage of the methods `batchUpdate(String sql, List<Object[]> batchArgs [,int[] argTypes])` is that an array with the data must be built up in advance. This is unfortunate if, for example, the data is in JavaBeans, which must then be read to copy it into an array.

Therefore, there is another way to get a `PreparedStatement` from Spring so that we can read our data from the data sources and transfer it to the `PreparedStatement`. The Spring data type `BatchPreparedStatementSetter` is used:[32]

```
package org.springframework.jdbc.core;

public interface BatchPreparedStatementSetter {
  int getBatchSize();
  void setValues(PreparedStatement ps, int i) throws SQLException;
}
```

---

[32] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/jdbc/core/BatchPreparedStatementSetter.html*

If a program uses this variant, the two methods must be implemented first. Next, the implementation will be passed to the `JdbcTemplate` method `int[] batchUpdate(String sql, BatchPreparedStatementSetter pss)`—currently there is only one method accepting `BatchPreparedStatementSetter`.

During execution, the first step Spring takes is to call the `getBatchSize()` method, which determines the number of rows to be processed. Next, Spring will call the `setValues(…)` method the same number of times. In our implementation of `setValues(…)`, we typically extract values from Java objects and set the corresponding states on the `PreparedStatement`. Additionally, Spring provides a counter value ranging from 0 to < `getBatchSize()`.

Here's a practical example of how this can be used:

```
record Photo(
  long profile, String name, boolean isProfilePhoto, LocalDateTime created
) { }
```

The record includes a profile ID, a name, a flag indicating whether it's a profile photo, and information about when the photo was created. This represents a single photo.

Now, let's consider a scenario where we need to save a list of photos:

```
List<Photo> photos = …;
```

The `BatchPreparedStatementSetter` can iterate over the list and set the column contents on the `PreparedStatement`:

```
BatchPreparedStatementSetter bpss = new BatchPreparedStatementSetter() {
  @Override public int getBatchSize() {
    return photos.size();
  }

  @Override
  public void setValues(PreparedStatement ps, int index)throws SQLException {
    ps.setLong( 1, photos.get( index ).profile() );
    ps.setString( 2, photos.get( index ).name() );
    …
  }
};
```

The inner anonymous class overrides both methods. The implementation of `getBatchSize()` directly returns the number of photos of the list. That's how much data to insert. `setValues(…)` gives us the `PreparedStatement` and the `index`. It's handy that Spring generates the values from `0` to `photos.size()` in an internal loop, so `photos.get(index)` simply extracts the `Photo`. From the `Photo`, our implementation must read the data and transfer it to the columns. The first place is the `profile` foreign key, the second place is the name of the photo, and so on. With `set*(…)`, the placeholders of the prepared statement are set.

The SQL string with the `INSERT` statement and the `BatchPreparedStatementSetter` is passed to `batchUpdate(…)`, and the query is executed:

```
var sql = """
    INSERT INTO photo (profile_fk, name, is_profile_photo, created)
    VALUES (?, ?, ?, ?)""";
int[] updateCounts = jdbcTemplate.batchUpdate( sql, bpss );
```

The list is processed internally by the `BatchPreparedStatementSetter`. This isn't visible to the caller from the outside.

AbstractInterruptibleBatchPreparedStatementSetter

The issue with `BatchPreparedStatementSetter` is that the number of elements must be known in advance, which isn't always the case. For example, suppose a program is given a CSV file containing data to be inserted into a database. In that case, it's impossible to know the number of rows in advance.

For this case, Spring declares the data type `AbstractInterruptibleBatchPreparedStatementSetter`[33] (see Figure 5.18).

The abstract class returns the value `Integer.MAX_VALUE` in `getBatchSize()`, so it's quasi infinite. Spring calls `isBatchExhausted(…)`; and if the implementation returns `false`, Spring terminates processing. To control a repetition, however, we don't override `isBatchExhausted(…)`, but a variant of `setValues(…)`, called `setValuesIfAvailable(…)`, which returns a `boolean` value.

To better understand the data type, a look at the source code helps:

```
public abstract class
  AbstractInterruptibleBatchPreparedStatementSetter
    implements InterruptibleBatchPreparedStatementSetter {

  private boolean exhausted;

  @Override
  public final void setValues(PreparedStatement ps,int i)throws SQLException{
    this.exhausted = ! setValuesIfAvailable( ps, i );
  }

  @Override public final boolean isBatchExhausted( int i ) {
    return this.exhausted;
```

---

[33] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/jdbc/core/support/AbstractInterruptibleBatchPreparedStatementSetter.html*

```
  }

  @Override public int getBatchSize() {
    return Integer.MAX_VALUE;
  }

  protected abstract boolean setValuesIfAvailable(
      PreparedStatement ps, int i ) throws SQLException;
}
```

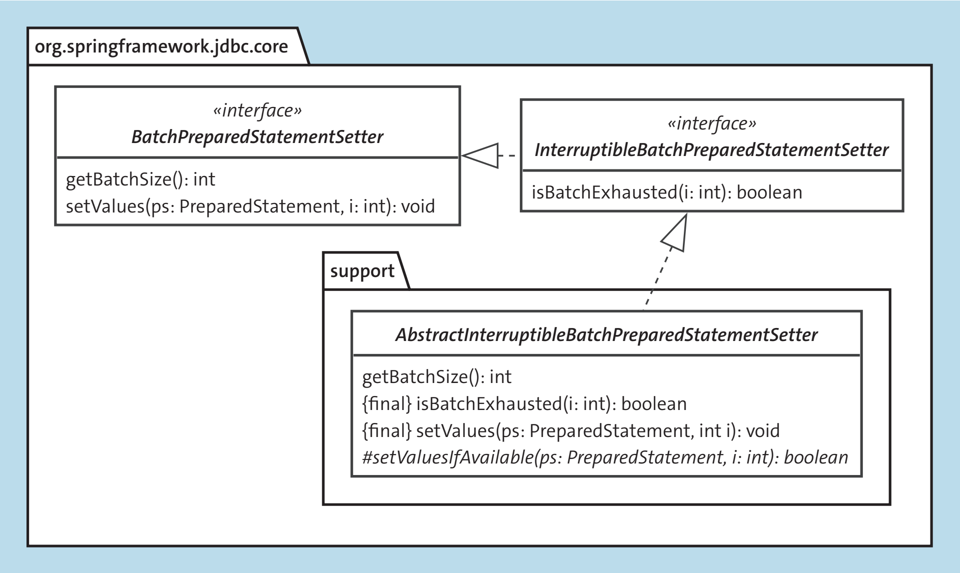

org.springframework.jdbc.core
«interface»
BatchPreparedStatementSetter
getBatchSize(): int
setValues(ps: PreparedStatement, i: int): void
«interface»
InterruptibleBatchPreparedStatementSetter
isBatchExhausted(i: int): boolean
support
AbstractInterruptibleBatchPreparedStatementSetter
getBatchSize(): int
{final} isBatchExhausted(i: int): boolean
{final} setValues(ps: PreparedStatement, int i): void
#setValuesIfAvailable(ps: PreparedStatement, i: int): boolean

Figure 5.18 “AbstractInterruptibleBatchPreparedStatementSetter”

The `setValues(…)` and `isBatchExhausted(…)` methods are marked as final, which means they can’t be overridden. Therefore, we only implement `setValuesIfAvailable(…)` and control whether the data provider has more data by returning `true` or `false`. If `setValuesIfAvailable(…)` returns `false`, the implementation sets a flag indicating that the batch is exhausted, which leads to an abort. The `isBatchExhausted(…)` method is provided by the `InterruptibleBatchPreparedStatementSetter` interface, which isn’t particularly interesting.

Consider the following code segment that accomplishes the same task as before, which is to save photos:

```
BatchPreparedStatementSetter bpss =
    new AbstractInterruptibleBatchPreparedStatementSetter() {

  @Override protected boolean
  setValuesIfAvailable( PreparedStatement ps, int index )
      throws SQLException {

    if ( index >= photos.size() ) return false;

    ps.setLong( 1, photos.get( index ).profile() );
    ps.setString( 2, photos.get( index ).name() );
    // …
    return true;
  }
};
```

The only method to override is `setValuesIfAvailable(…)`. The visibility remains `protected`, and it's not useful to increase the visibility. In the method, Spring passes a `PreparedStatement` and the index from `0` to `Integer.MAX_VALUE`. We need to indicate via a `boolean` if there are more elements. Because Spring increments the index for us, we return `false` if `index >= photos.size()` because then batch processing completes. If there are elements, `set*(…)` sets the values, and the `setValuesIfAvailable(…)` method returns `true` for the next run.

ParameterizedPreparedStatementSetter

What is special about the `BatchPreparedStatementSetter` is that the Spring Framework calls us the callback methods with an index in a loop. This is always handy when it comes to index-based accesses, such as with an array or a `List`, where the index maps to the object. However, you could go one step further and have the object given directly instead of the index when Spring iterates over the data structure. The following `JdbcTemplate` method is used for this:

```
int[][] batchUpdate(String sql, Collection<T> batchArgs,
int batchSize, ParameterizedPreparedStatementSetter<T> pss)
```

A `ParameterizedPreparedStatementSetter` is passed, which declares a method:

```
void setValues(PreparedStatement ps, T argument)
```

A `PreparedStatement` is passed along with each element of the `Collection`. It's the responsibility of the implementation to transfer the states from the object to the `PreparedStatement`.

Let's stick with the example of saving `Photo` objects:

```
Collection<Photo> photos = …;
var sql = """
    INSERT INTO photo (profile_fk, name, is_profile_photo, created)
    VALUES (?, ?, ?, ?)""";
```

```
int[][] ints = jdbcTemplate.batchUpdate(sql, photos, 10, (ps, photo)->{
  ps.setLong( 1, photo.profile() );
  ps.setString( 2, photo.name() );
  // …
} );
```

Four arguments are passed to the `batchUpdate(…)` method. The SQL statement and the collection are at the beginning, followed by the batch size. This isn't the total number of all elements, but the "chunk size", that is, how many elements should be processed in one batch. Our batch size is `10`, but a benchmark must show which size is optimal.

The last argument is the `ParameterizedPreparedStatementSetter` implementation, and because the type is a functional interface, a lambda expression results in compact code. If the photos are present as a `Collection`, this notation is the shortest overall.

5.7.3 Batch Methods at NamedParameterJdbcTemplate

Compared to the `NamedParameterJdbcTemplate`, the `JdbcTemplate` has a relatively large number of batch update methods. The `NamedParameterJdbcTemplate`, on the other hand, offers only two `batchUpdate(…)` methods:

- `int[] batchUpdate(String sql, Map<String,?>[] batchValues)`
- `int[] batchUpdate(String sql, SqlParameterSource[] batchArgs)`

For the methods, the SQL string is passed first again. Next, the named parameters must be determined. The overloaded methods give two options: use an array of maps or use an array of `SqlParameterSource` objects. Because multiple rows are processed in a batch, they are kept in an array; there is no variant for `List` or `Collection`.

Note

The `NamedParameterJdbcTemplate` is unable to handle `BatchPreparedStatementSetter` and its subtypes, which indicates that certain operations still need to be performed using the `JdbcTemplate`.

Returning to our list of `Photo` objects to be saved, using `NamedParameterJdbcTemplate` replaces the question marks found in a regular `PreparedStatement` with named parameters within the SQL statement:

```
record Photo(
  long profile, String name, boolean isProfilePhoto, LocalDateTime created
) { }

List<Photo> photos = …
```

```
MapSqlParameterSource[] args = photos.stream().map(
        photo -> new MapSqlParameterSource()
            .addValue( "profile", photo.profile() )
            .addValue( "name", photo.name() )
            .addValue( "is_profile_photo", photo.isProfilePhoto() )
            .addValue( "created", photo.created() ) )
    .toArray( MapSqlParameterSource[]::new );

var sql = """
    INSERT INTO photo ( profile_fk, name, is_profile_photo, created )
    VALUES (:profile, :name, :is_profile_photo, :created)""";
int[] ints = namedJdbcTemplate.batchUpdate( sql, args );
```

The `batchUpdate(…)` method gets the batch values from an array of `MapSqlParameterSource` objects. This array is built via a `Stream` expression.

5.7.4 SqlParameterSourceUtils

In the example, we created a `MapSqlParameterSource` object manually and filled it with the data from the `Photo` object. This is a step to create an array of `MapSqlParameterSource` objects from a collection of objects.

Spring offers the class `SqlParameterSourceUtils`[34] to convert data into `SqlParameterSource` arrays. Three selected static methods of the class are as follows:

- `SqlParameterSource[] createBatch(Object... candidates)`
- `SqlParameterSource[] createBatch(Collection<?> candidates)`
- `SqlParameterSource[] createBatch(Map<String,?>[] valueMaps)`

The objects provided with `Object...` and `Collection<?>` are JavaBeans that are transferred to `SqlParameterSource` objects via reflection. In the last case, the contents of an array of maps is transferred to an array of `SqlParameterSource` objects.

5.7.5 Configuration Properties

There are some properties that control the behavior when fetching data. The settings can be made either in code via `JdbcTemplate` (there are setters/getters) or externally via configuration properties:

---

[34] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/jdbc/core/namedparam/SqlParameterSourceUtils.html*

Property of JdbcTemplate	Configuration Property	Meaning
`fetchSize`	`spring.jdbc.template.fetch-size`	The number of lines that will be loaded into the buffer. At `-1`, it's the default from the JDBC driver.
`maxRows`	`spring.jdbc.template.max-rows`	The maximum number of lines. At `-1`, it's the default from the JDBC driver.
`queryTimeout`	`spring.jdbc.template.query-timeout`	The query timeout. Applies to the JDBC driver. Suffixes are allowed; without a suffix, the unit is seconds.

Table 5.1 Setting Options for the "JdbcTemplate"

Now, let's take a closer look at each of the three settings. The `fetchSize` setting determines the amount of data that is loaded in a single fetch, which refers to the communication between the Java application and the database via the JDBC driver. If a program retrieves millions of rows using an SQL statement and iterates through each row using `next()`, `next()`, `next()`..., the JDBC driver won't fetch each row individually from the database. Instead, it will bundle them and store them in an internal buffer. The number of rows loaded is determined by the fetch size, which can be compared to the buffer size for `BufferReader` or `BufferedInputStream`.

The `maxRows` setting is useful for protecting against wrongly formulated SQL statements that could potentially retrieve all data. In such cases, `maxRows` will force the client to paginate the results because the program can't read more than the specified number of rows.

Lastly, a `queryTimeout` can be set if the database doesn't respond or if a program gets stuck in a transaction for an extended period. This is because JDBC has a blocking API, and a `queryTimeout` ensures that the program doesn't wait indefinitely for a response.

5.8 BLOBs and CLOBs *

Apart from the standard JDBC scalar types such as `VARCHAR` and `NUMERIC`, databases can also store larger objects known as *large objects* (LOBs). LOBs are further classified into two types: *binary large objects* (BLOBs) and *character large objects* (CLOBs).

With the `JdbcTemplate`, Spring can handle these LOBs as well. There are two ways to handle LOBs in Spring: using `SqlLobValue` and using a `LobHandler` with an `AbstractLobStreamingResultSetExtractor`.

5.8.1 SqlLobValue

To represent large datasets, the Spring Framework declares the `SqlLobValue` data type[35] (see Figure 5.19).

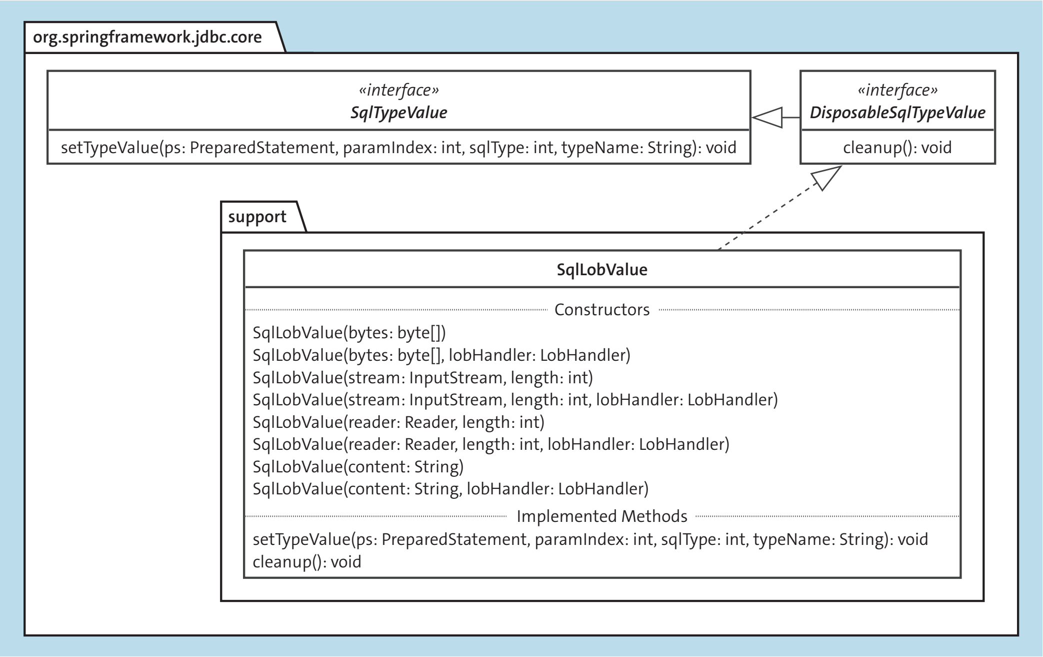

org.springframework.jdbc.core
«interface»
SqlTypeValue
setTypeValue(ps: PreparedStatement, paramIndex: int, sqlType: int, typeName: String): void
«interface»
DisposableSqlTypeValue
cleanup(): void
support
SqlLobValue
Constructors
SqlLobValue(bytes: byte[])
SqlLobValue(bytes: byte[], lobHandler: LobHandler)
SqlLobValue(stream: InputStream, length: int)
SqlLobValue(stream: InputStream, length: int, lobHandler: LobHandler)
SqlLobValue(reader: Reader, length: int)
SqlLobValue(reader: Reader, length: int, lobHandler: LobHandler)
SqlLobValue(content: String)
SqlLobValue(content: String, lobHandler: LobHandler)
Implemented Methods
setTypeValue(ps: PreparedStatement, paramIndex: int, sqlType: int, typeName: String): void
cleanup(): void

Figure 5.19 Methods and Type Relationships of "SqlLobValue"

The `SqlLobValue` class declares various constructors to hold the data. Sources include a byte array, an `InputStream`, a `Reader`, and a `String`. The `SqlLobValue` type then represents the data. The method `setTypeValue(…)` sets the content into the column via a `PreparedStatement`.

Write LOBs

Here's an example demonstrating how to use `SqlLobValue`. Suppose we have an object array with values for three columns. In the first column, a regular string should be placed behind the variable `name`; in the second column, an `SqlLobValue` should represent an LOB `stream` with content coming from an IO stream; and in the third column, an `SqlLobValue` should come from a `text`:

```
var name = …, var stream = …, var len = …, var text = …;
Object[] args = {
    name,
```

---

[35] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/jdbc/core/support/SqlLobValue.html*

```
    new SqlLobValue( stream, len ),
    new SqlLobValue( text )
};
```

The `update(…)` method of `JdbcTemplate` can write the data. For this, the method gets the values from `SqlLobValue`, transfers them to the respective columns, and fills the LOB in the database this way:

```
jdbcTemplate.update(
    "INSERT INTO Table (col1, col2, col3) VALUES (?, ?, ?)",
    args,
    new int[]{ Types.VARCHAR, Types.BLOB, Types.CLOB }
);
```

The `update(…)` method is passed three arguments: the SQL string, the data to insert, and the type information.

Next, let's take a quick look at two other Spring data types before we proceed with reading LOBs from the database.

5.8.2 LobHandler and DefaultLobHandler

With data type `LobHandler`,[36] you can primarily read LOBs from a `ResultSet`. A LOB can also be written via `LobCreator`. An implementation of `LobHandler` is `DefaultLobHandler`.[37]

Reading a LOB isn't a complicated process. Here is the source code for `getBlobAsBinaryStream(…)` in `DefaultLobHandler`:

```
public InputStream getBlobAsBinaryStream( ResultSet rs, int columnIndex )
    throws SQLException {
  logger.debug( "Returning BLOB as binary stream" );
  if ( this.wrapAsLob ) {
    Blob blob = rs.getBlob( columnIndex );
    return blob.getBinaryStream();
  }
  else {
    return rs.getBinaryStream( columnIndex );
  }
}
```

Listing 5.5 https://github.com/spring-projects/spring-framework/blob/main/spring-jdbc/src/main/java/org/springframework/jdbc/support/lob/DefaultLobHandler.java

---

[36] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/jdbc/support/lob/LobHandler.html*

[37] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/jdbc/support/lob/DefaultLobHandler.html*

When passing the `ResultSet`, the `columnIndex` is also passed. By default, the flag `wrapAsLob` is `false`, which means that the `getBlobAsBinaryStream(…)` method directly forwards to the `ResultSet`. However, the support for the `java.sql.Blob` and `java.sql.Clob` types in JDBC drivers can vary, and, in some cases, it may be advantageous to use them. The Javadoc for `LobHandler` provides more detailed information on this topic.

Pass LobHandler at SqlLobValue

The `SqlLobValue` container represents data that will later be written to a `ResultSet`. Here's a quick peek inside the constructors of `SqlLobValue`:

```
class SqlLobValue … {

  public SqlLobValue(@Nullable String content) {
    this(content, new DefaultLobHandler());
  }
  public SqlLobValue(@Nullable String content, LobHandler lobHandler) { … }
  …
}
```

All constructors are overloaded, not just the data types. There is always a version that accepts the source and a second constructor where you can pass your own `LobHandler`. The simplified method without `LobHandler` always builds a `DefaultLobHandler` internally and works with it. (This also means that if we make many calls via `SqlLobValue`, a new `DefaultLobHandler` will then always be built).

So, our previous example can be rewritten like this:

```
LobHandler lobHandler = new DefaultLobHandler();

Object[] args = {
    name,
    new SqlLobValue( stream, len, lobHandler ),
    new SqlLobValue( description, lobHandler )
};
```

This saves multiple instantiations within the constructors.

5.8.3 Read LOBs via AbstractLobStreamingResultSetExtractor

One way to read LOBs is to call one of the `get*(…)` methods on the `LobHandler`. There is another way via a special `ResultSetExtractor` (the type described in Section 5.5.3), but it requires more code.

Consider this hypothetical example where we have an image stored as a LOB in a column named `col`, and we also have a criterion for selecting a column:

```
var sql = "SELECT col FROM Table WHERE id=?";
```

On the `JdbcTemplate`, the `query(…)` method can be called with the SQL, a `ResultSetExtractor`, and an assignment of the question mark:

```
ResultSetExtractor<Void> rse =
    new AbstractLobStreamingResultSetExtractor<>() {

  private final LobHandler lobHandler = new DefaultLobHandler();

  @Override
  public void streamData( ResultSet rs ) throws SQLException {
    InputStream is = lobHandler.getBlobAsBinaryStream( rs, "col" );
    …
  }

};
jdbcTemplate.query( sql, rse, id );
```

Remember, `ResultSetExtractor` is a data type that a program always uses when the program itself iterates over the `ResultSet` and wants to return a result. However, by specifying `Void` in angle brackets, that is, `ResultSetExtractor<Void>`, the program says that nothing is returned because the processing takes place inside the `ResultSetExtractor` and nothing comes outside.

The program uses the notation for inner anonymous classes to create a subclass from the abstract class `AbstractLobStreamingResultSetExtractor`[38] and to implement the abstract method `streamData(ResultSet rs)`. Using the `LobHandler`, `getBlobAsBinaryStream(…)` returns an `InputStream` to the LOB. We can consume the `InputStream` and process the data.

5.9 Subpackage org.springframework.jdbc.core.simple *

The `org.springframework.jdbc.core` package has quite a few subpackages, one of which is `org.springframework.jdbc.core.simple`. At its core, the package consists of three classes (see Figure 5.20):

---

[38] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/jdbc/core/support/AbstractLobStreamingResultSetExtractor.html*

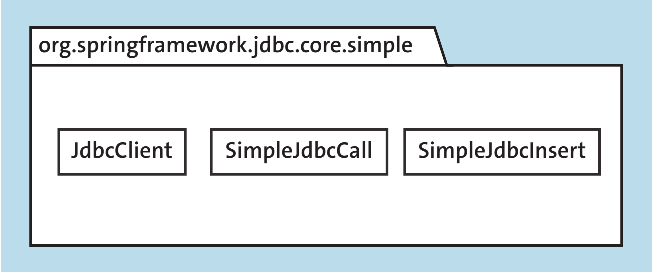


Figure 5.20 Three Central Classes in "org.springframework.jdbc.core.simple"

- The `JdbcClient` has a fluent API and helps to run JDBC queries and updates with ease. As it currently stands, the class will make its way into Spring Framework 6.1 (as of September 2023).
- `SimpleJdbcCall` for calling stored procedures and `SimpleJdbcInsert` for data insertion.

5.9.1 SimpleJdbcInsert

By accessing the metadata of the JDBC driver internally, `SimpleJdbcInsert` can save us some effort. For instance, we don't need to specify column names for insertion when using `SimpleJdbcInsert`, as it retrieves the column names from the metadata of the corresponding table.

The following repository `LikesInserter` has the task of connecting two profiles:

```
@Repository
class LikesInserter {
  private final SimpleJdbcInsert jdbcInsertLikes;

  LikesInserter( DataSource dataSource ) {
    jdbcInsertLikes = new SimpleJdbcInsert( dataSource )
                          .withTableName( "Likes" );
  }
  public void addLikes( long liker, long likee ) {
    jdbcInsertLikes.execute( Map.of( "liker_fk", liker,
                                     "likee_fk", likee ) );
  }
}
```

The first step is to pass a `DataSource` to the constructor of `SimpleJdbcInsert`.[39] Next, the table name is set using `withTableName(…)`, which in our scenario is `Likes`. Although the class offers `withCatalogName(…)` and `withSchemaName(…)` methods for more control, it doesn't have many methods in general. By following the specifications, the class can access the metadata of the `Likes` table and determine its columns. `JdbcInsertLikes` saves the instance internally.

---

[39] *https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/jdbc/core/simple/SimpleJdbcInsert.html*

The repository `LikesInserter` can be injected, and then you can call the method `addLikes(…)`. Two instances of `long` are passed, which are the IDs of the profiles that like each other. To send the SQL `INSERT`, the `execute(…)` method is used. There are different parameterizations for it—here, the one with `Map` was chosen. Two key-value pairs are passed to the associative data structure, making a total of four values. The key is the name of the column, and the associated value is the parameter assignment.

SimpleJdbcInsert and Generated Keys

The `SimpleJdbcInsert` class can do a bit more, such as return automatically generated keys. Four `executeAndReturnKey*(…)` methods are used for this purpose. The returns are as follows:

- `Number` (base class of the wrapper classes and `BigInteger`) for numeric values
- `KeyHolder` for strings and composite keys

SimpleJdbcInsert and Batch Updates

`SimpleJdbcInsert` also has two batch methods. They differ in how the parameters are passed:

- `int[] executeBatch(Map<String,?>... batch)`
- `int[] executeBatch(SqlParameterSource... batch)`

A batch is a collection of individual operations. These can modify a different number of rows. The `int` array encodes the number of modified rows for each single step in the batch. For example, suppose we want to insert rows into the `Photo` table (a profile has several photos):

```
Map<String, Object> map1 = Map.of( "profile_fk", …,
                                   "name", …,
                                   "is_profile_photo", …,
                                   "created", … );
Map<String, Object> map2 = …
photoJdbcInsert.executeBatch( map1, map2 );
```

The column names are stored in a `Map` and associated with their respective values. These maps are then passed to the `executeBatch(…)` method as varargs.

Note

A small disadvantage is that the `executeBatch(…)` methods can't return automatically generated keys. However, we have this disadvantage elsewhere as well.

5.10 Package org.springframework.jdbc.object *

The one remaining Spring JDBC package to discuss is `org.springframework.jdbc.object`. This package contains data types that represent SQL operations as objects so that they can be used to make uncomplicated queries or changes to the database. The base type is `RdbmsOperation` and that's why we often talk about *RDBMS objects*.

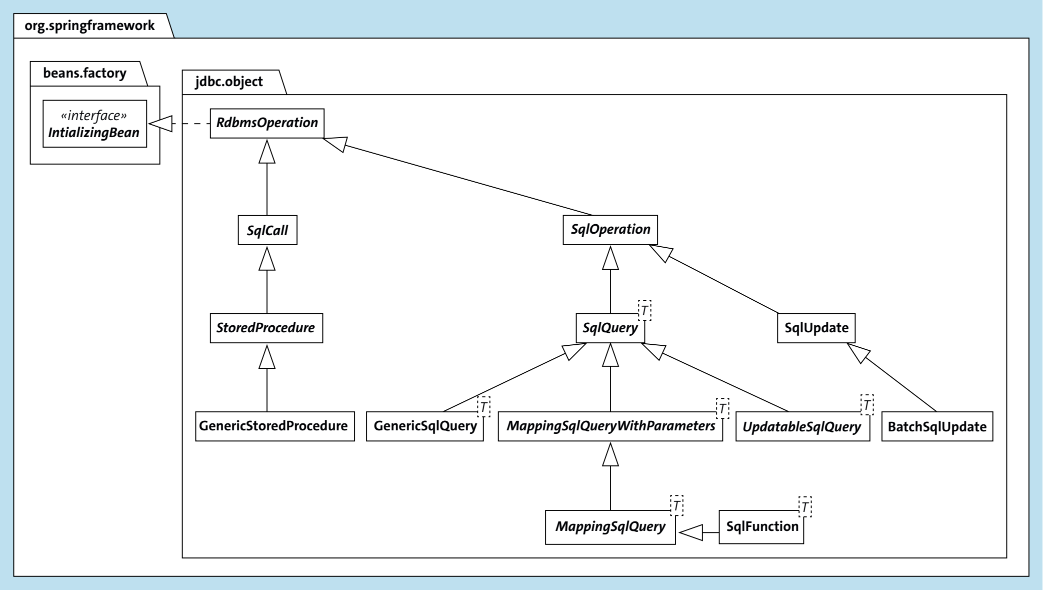

org.springframework
beans.factory
«interface»
IntializingBean
jdbc.object
RdbmsOperation
SqlCall
SqlOperation
StoredProcedure
SqlQuery
T
SqlUpdate
GenericStoredProcedure
GenericSqlQuery
T
MappingSqlQueryWithParameters
T
UpdatableSqlQuery
T
BatchSqlUpdate
MappingSqlQuery
T
SqlFunction
T

Figure 5.21 Data Types from "org.springframework.jdbc.object"

The Unified Modeling Language (UML) diagram in Figure 5.21 shows numerous types, but in practice, you're dealing with only a few data types. In modern Spring applications, the types from `org.springframework.jdbc` are hardly to be found, so it's only discussed briefly here.

5.10.1 MappingSqlQuery

An object of the type `MappingSqlQuery`[40] internally manages the following:

- `DataSource`
- SQL string for a query
- Implementation for mapping a row to an object in the form of a `mapRow(…)` method

The object consequently encapsulates various states, and the `MappingSqlQuery` object only needs to be given the parameters via a vararg when it's activated. After that, the result can be collected.

---

[40] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/jdbc/object/MappingSqlQuery.html*

Form a Subclass of MappingSqlQuery

Here’s an example that retrieves `Photo` objects based on two search parameters. A `@Bean` method will put a `MappingSqlQuery<Photo>` in the context:

```
@Configuration( proxyBeanMethods = false )
class PhotoMappingSqlQuery {

  @Bean
  public MappingSqlQuery<Photo> photoMppingSqlQuery( DataSource dataSource ) {
    MappingSqlQuery<Photo> sqlQuery = new MappingSqlQuery<>( dataSource, """
        SELECT id, profile_fk, name, is_profile_photo, created
        FROM Photo
        WHERE is_profile_photo = ? AND created > ?""" ) {

      @Override
      protected Photo mapRow( ResultSet rs, int __ ) throws SQLException {
        return new Photo(
            rs.getLong( "id" ), rs.getLong( "profile_fk" ),
            rs.getString( "name" ), rs.getBoolean( "is_profile_photo" ),
            rs.getTimestamp( "created" ).toLocalDateTime() );
      }
    };
    sqlQuery.declareParameter( new SqlParameter
( "is_profile_photo", BOOLEAN) );
    sqlQuery.declareParameter( new SqlParameter( "created", TIMESTAMP ) );
    return sqlQuery;
  }
}
```

`MappingSqlQuery` is an abstract class with one abstract method:

```
protected abstract T mapRow(ResultSet rs, int rowNum)
```

Therefore, a subclass must override the method. In addition, there are two constructors, one of which must call the subclass:

- `MappingSqlQuery()`
- `MappingSqlQuery(DataSource ds, String sql)`

The example uses an inner anonymous class for the implementation and calls the parameterized constructor with the `DataSource` and the SQL statement. The `SELECT` selects all columns from the `Photo` table, and two `?` indicate parameterization: callers should control whether a `Photo` is a profile photo and whether it was created after a certain time.

The abstract method `mapRow(…)` looks familiar from `org.springframework.jdbc.core.RowMapper`—the type was described in Section 5.5.1. However, the `MappingSqlQuery` class doesn’t implement `RowMapper`: the method would otherwise have had to be `public`, but that wasn't desired by the design. `mapRow(…)` gets a `ResultSet` of a row, and then the program reads the row and transfers the columns to a `Photo` object.

Then, the program sets the prepared statement parameters of the `MappingSqlQuery` via `declareParameter(…)`. `SqlParameter` objects are passed that determine two pieces of information: the name of the column and the SQL type. `is_profile_photo` is a `boolean`, and `created` is a `Timestamp`.

Using a MappingSqlQuery Object

Elsewhere, a `MappingSqlQuery` object can be injected typed with `Photo`:

```
@Autowired MappingSqlQuery<Photo> sqlQuery;
```

Subsequently, methods such as `execute(…)` can be submitted for execution:

```
sqlQuery.execute( true, LocalDateTime.MIN ).forEach( System.out::println );
```

It's practical to specify the parameters via a vararg. The result is a `List` that can be traversed by `forEach(…)`.

A `MappingSqlQuery` object can be further parameterized, for example, with the fetch size or the maximum number of rows:

```
sqlQuery.setFetchSize( 100 );
sqlQuery.setMaxRows( 100 );
```

5.11 Transactions

Transactions play a critical role in ensuring the consistency and integrity of data in databases. As a result, Spring provides robust support for managing transactions in database applications. In this section, we'll delve into the details of transaction management in Spring and explore the different ways in which Spring makes it easier to start, commit, and roll back transactions.

5.11.1 ACID Principle

A transaction in a database involves a group of operations that are combined into a single unit that the RDBMS must process consistently. The ACID principle is essential in this process and encompasses four key elements: *atomicity*, *consistency*, *isolation*, and *durability*:

- **Atomicity**
  This refers to the all-or-nothing principle, where either all operations are executed and a commit follows, or a rollback occurs in case of an error, returning the database to its initial state. This ensures that the database remains in a consistent state and prevents any partial updates that may result in data corruption.

- **Consistency**
  This ensures that the data remains accurate and valid before, during, and after the transaction. Integrity conditions, such as primary key constraints, must be enforced, and any errors must be rectified before committing the transaction. Any invalid data that results from a failed transaction should be rolled back to the previous state.
- **Isolation**
  This is about the demarcation of operations in a transaction. When writing or reading data, these operations must be separate from other transactions to avoid data corruption. Although isolation is often relaxed for performance reasons, it remains crucial in ensuring that each transaction operates on consistent data.
- **Durability**
  This is the guarantee that data will be stored reliably and persistently, even in the event of system failure. Additionally, the consistency of the data must not be compromised even in the face of system failure, which makes durability closely linked to consistency.

5.11.2 Local or Global/Distributed Transactions

When we combine operations into a transaction, we must have a data source that basically allows commit and rollback. Full RDBMSs are inherently capable of this. Because there are NoSQL databases, messaging systems, and so on in addition to database management systems, we generally refer to this as a *resource provider*.

If only a single database (or resource provider) is used for transaction processing, we refer to *local transactions*. In Java, local transactions are possible with a database via `java.sql.Connection`; the JDBC driver uses the capabilities of the database. A Java program can roll back or commit via the `Connection` interface.

In addition to local transactions, there are *distributed transactions*, which are also called *global transactions*. In this case, the data originates from several resource providers (e.g., several databases or a messaging system and a database) and must be coordinated. In the best case, the ACID criteria also apply to global transactions, but then a transaction manager is needed to bring the different resources together. A transaction manager is a component of an RDBMS, and we want to look at how we can start, confirm, or rollback a transaction locally.

5.11.3 JDBC Transactions: Auto-Commit

For performance reasons, transactions should be as short as possible. Therefore, the *auto-commit mode* is automatically set for a JDBC connection. In this mode, each SQL statement is *committed* after execution. This means that we never have multiple operations combined into one transaction in standard mode, but when an operation is sent and is successful, it’s automatically followed by a commit. If there is another query over the same connection, this is again completed with a commit, and so on.

If a program wants to combine several operations in one transaction, the auto-commit mode must first be switched off. To do this, `setAutoCommit(false)` is called on the `Connection` object. After the call, the transaction starts. We initiate the end of a transaction by calling the connection method `commit()` or `rollback()`.

Transaction with JDBC Connection

To perform multiple operations in a transactional context using a JDBC `Connection`, you can turn off auto-commit mode and enclose the code in a `try-catch` block. Here's an example:

```
connection.setAutoCommit( false );

try {
  … INSERT …
  … UPDATE …
  connection.commit();
}
catch ( Throwable e ) {
  connection.rollback();
  // throw e;
}
```

The transaction is committed, and all data is written if the `try` block executes without any errors due to the business logic. However, if an exception occurs, the transaction can't be left incomplete. Therefore, errors are caught, and a rollback is performed on the `Connection`. Depending on the task, a `Throwable` may be caught or passed up in different programming approaches.

5.11.4 PlatformTransactionManager

Besides RDBMS, there are other resource providers—they will have a different API to start, commit, or abort a transaction. That's why Spring introduced its own abstraction in the form of the data type `PlatformTransactionManager`.[41] This has three core methods: `commit(…)`, `rollback(…)`, and `getTransaction(…)` (see Figure 5.22).

The basic idea is that the previous `commit()` and `rollback()` methods on the JDBC `Connection` move toward `PlatformTransactionManager`. The `PlatformTransactionManager` interface must be implemented by a resource provider or transaction manager. The Spring Framework offers different implementations. For example, the UML diagram shows the `DataSourceTransactionManager`, which is initialized with a `DataSource`. A subclass of `DataSourceTransactionManager` is `JdbcTransactionManager`. If `commit(…)` or `roll-`

---

[41] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/transaction/PlatformTransactionManager.html*

`back(…)` is then called on this particular implementation, `JdbcTransactionManager` forwards the methods to the JDBC `Connection`.

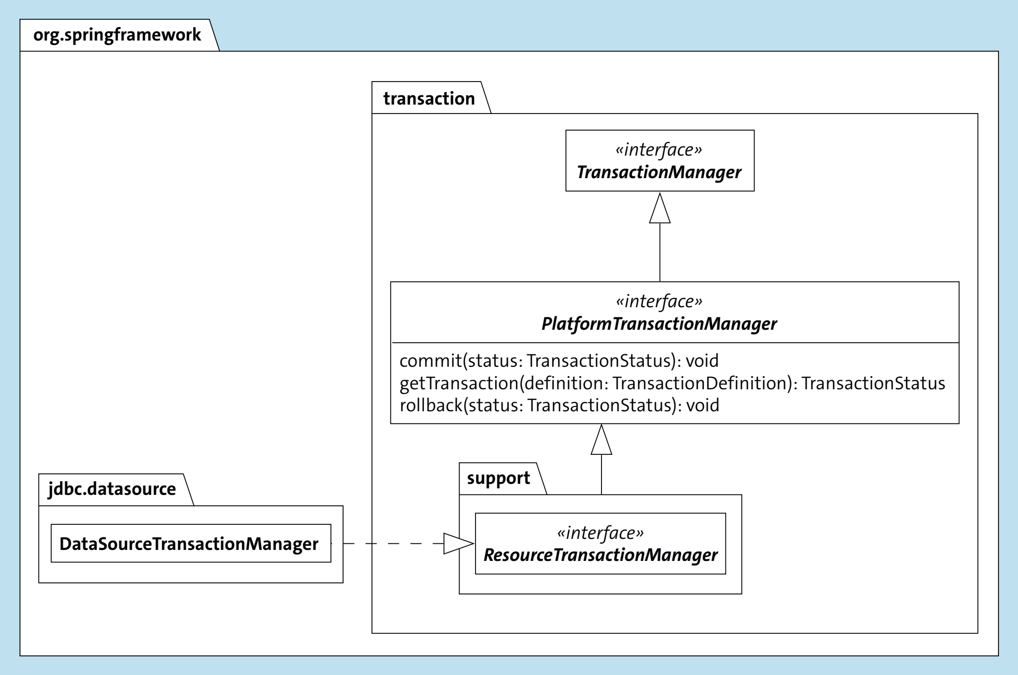

org.springframework
transaction
«interface»
TransactionManager
«interface»
PlatformTransactionManager
commit(status: TransactionStatus): void
getTransaction(definition: TransactionDefinition): TransactionStatus
rollback(status: TransactionStatus): void
support
«interface»
ResourceTransactionManager
jdbc.datasource
DataSourceTransactionManager

Figure 5.22 "PlatformTransactionManager" Methods and Type Relationships

Spring Boot automatically builds a `PlatformTransactionManager` via auto-configuration for an existing database connection; a client can get this injected:

```
@Autowired PlatformTransactionManager transactionManager;
```

The usage in a program looks like this:

```
TransactionDefinition def = null;
TransactionStatus status = transactionManager.getTransaction( def );

try {
  …
  transactionManager.commit( status );
}
catch ( Throwable e ) {
  transactionManager.rollback( status );
  // throw e;
}
```

The code is similar to the JDBC `Connection`, but the API is a bit more complex because `commit(…)` and `rollback(…)` require a `TransactionStatus` object to be passed.

Open Connection during a Transaction

In large applications, the code of a transaction may be spread over several classes and methods. The challenge here is to "transport" the active transaction from one place to another. Therefore, we want to put `PlatformTransactionManager` into a larger context and look at the possibilities with which we can get a JDBC `Connection` and what the consequences are.

In principle, there are two ways to open connections: one way is to have a `DataSource` injected and then call `getConnection()`. After opening a `Connection`, a program must close this connection at the end. Here's an example that creates two connections and checks if they are the same or different:

```
{
  Connection con1 = dataSource.getConnection();
  log.info( "{}", System.identityHashCode( con1 ) );     // 462432055
  con1.close();

  Connection con2 = dataSource.getConnection();
  log.info( "{}", System.identityHashCode( con2 ) );     // 717246890
  con2.close();
}
```

Two different connections are displayed. The call `System.identityHashCode(…)` returns a number. Two different numbers tell us that we're dealing with two different objects—we leave collisions out of it.[42]

The second possibility is provided by the class `DataSourceUtils`[43] and the static method `getConnection(DataSource)`—the class we mentioned in Section 5.3.10:

```
{
  Connection con1 = DataSourceUtils.getConnection( dataSource );
  log.info( "{}", System.identityHashCode( con1 ) );     // 469707537
  DataSourceUtils.releaseConnection( con1, dataSource );

  Connection con2 = DataSourceUtils.getConnection( dataSource );
  log.info( "{}", System.identityHashCode( con2 ) );     // 388244568
  DataSourceUtils.releaseConnection( con2, dataSource );
}
```

---

[42] Calling `hashCode()` on `Connection` itself runs the risk of `Connection` overriding the `hashCode()` method and giving us something we can't use for object identity comparison.

[43] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/jdbc/datasource/DataSourceUtils.html*

To release the connection, we call `releaseConnection(…)`, and the `Connection` and the `DataSource` are passed. These are two different numbers, so again two new connections.

We want to rewrite the code a bit and bring in the transaction manager:

```
TransactionStatus status = transactionManager.getTransaction( null );

try {
  Connection con1 = DataSourceUtils.getConnection( dataSource );
  log.info( "{}", System.identityHashCode( con1 ) );      // 1234567890
  // vs. Connection con1 = dataSource.getConnection();
  // log.info( "{}", System.identityHashCode( con1 ) );   // 1212121212
  DataSourceUtils.releaseConnection( con1, dataSource );
  // vs. JdbcUtils.closeConnection( con1 );

  Connection con2 = DataSourceUtils.getConnection( dataSource );
  log.info( "{}", System.identityHashCode( con2 ) );      // 1234567890
  // vs. Connection con2 = dataSource.getConnection();
  // log.info( "{}", System.identityHashCode( con2 ) );   // 8989898989
  DataSourceUtils.releaseConnection( con2, dataSource );
  // vs. JdbcUtils.closeConnection( con2 );

  transactionManager.commit( status );
}
catch ( Throwable e ) {
  transactionManager.rollback( status );
}
```

There is an important difference between the `getConnection(…)` methods:

- `getConnection()` of `DataSource`
  A new connection is always established.[44]
- `getConnection(…)` of `DataSourceUtils`
  An existing connection due to a transaction will be continued.

Let's examine the process in greater depth. Invoking the `getTransaction()` method on the transaction manager initializes the transaction, establishes a connection, and sets the auto-commit feature to `false`. This JDBC connection is stored internally in a thread-local variable, which belongs to the current thread. In other words, the `Connection` for the transaction is associated with the thread that invokes `getTransaction()`.

When the `DataSourceUtils.getConnection()` method is called, it retrieves the `Connection` from the thread-local variable. This allows `DataSourceUtils.getConnection()` and `DataSourceTransactionManager` to share the `Connection`. Because the `Connection` is associated with the thread, it enables different methods within the same thread to access the `Connection` using `DataSourceUtils.getConnection()`, release it with `releaseConnection()`, and participate in the same transaction.

---

[44] At least logically, we ignore any connection pools.

The `releaseConnection()` method is aptly named because the `Connection` isn't closed; it remains open for the transaction. At the end of the transaction, the `commit()` method is called on the transaction manager, not directly on the `Connection`.

`JdbcTemplate`, as well as other classes such as `NamedParameterJdbcTemplate`, request `Connections` from `DataSourceUtils.getConnection()`. As a result, these types don't include transaction methods. `PlatformTransactionManager` is responsible for transactional control.

Storing the `Connection` in a thread-local variable has been effective in the past, but it has a drawback. By default, only one `Connection` per `DataSource` and thread is allowed. If a new thread attempts to access the `Connection`, additional assistance is required.

5.11.5 TransactionTemplate

Program code, such as for starting a transaction at the beginning and ending it at the end, can be simplified using an *execute-around idiom*. With this idiom, your own program code is placed in the body, so to speak, and an operation is performed around it (*execute around*). In our case, the block is our own code, and starting and ending the transaction represent the "around."

The Spring Framework provides a convenient way to implement the execute-around idiom for transactions using the `TransactionTemplate`.[45] This template allows developers to execute a block of code within a transaction, simplifying the process of starting and ending transactions.

One of the benefits of using `TransactionTemplate` is that it's automatically initialized with a `PlatformTransactionManager` by the auto-configuration process. This is necessary because the `TransactionTemplate` needs to call methods such as `commit()` and `rollback()` to manage the transaction.

We can have the instance injected:

```
@Autowired TransactionTemplate transactionTemplate;
```

Central to `TransactionTemplate` is the `execute(TransactionCallback<T> action)` method. With `execute(…)`, we pass our program block via the callback interface `TransactionCallback`[46]. The declaration is as follows:

---

[45] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/transaction/support/TransactionTemplate.html*

[46] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/transaction/support/TransactionCallback.html*

```
interface TransactionCallback<T> {
  T doInTransaction(TransactionStatus status)
}
```

Example for execute(TransactionCallback)

To better understand how to use the `execute(…)` method, let's consider an example. Suppose we need to execute two SQL `UPDATE` statements within a transactional context:

```
var result = transactionTemplate.execute( transactionStatus -> {
  …
  jdbcTemplate.update( … );
  jdbcTemplate.update( … );
  …
  return …
} );
```

TransactionCallbackWithoutResult

The `TransactionCallback` type can return a result for `excecute(…)`, but no return may be necessary. Then, a subclass of `TransactionCallbackWithoutResult`[47] can be created (see Figure 5.23). This class implements the `TransactionCallback` interface, and we override `doInTransactionWithoutResult()`.

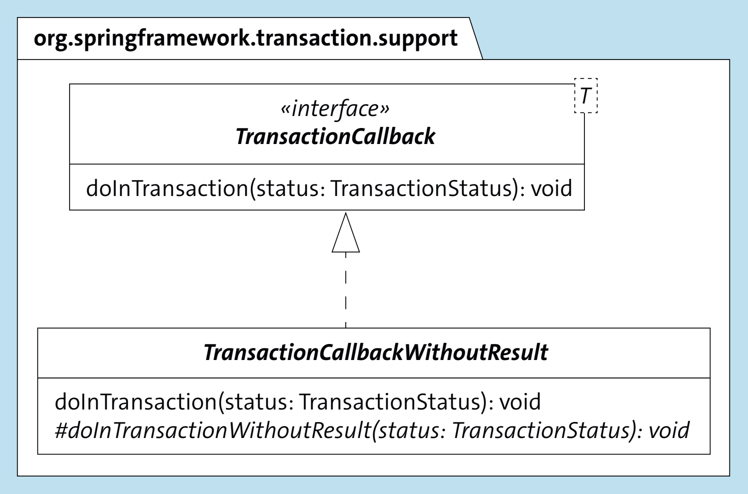

org.springframework.transaction.support
«interface»
TransactionCallback
T
doInTransaction(status: TransactionStatus): void
TransactionCallbackWithoutResult
doInTransaction(status: TransactionStatus): void
#doInTransactionWithoutResult(status: TransactionStatus): void

Figure 5.23 "TransactionCallbackWithoutResult": Returns No Result

Here you have to consider whether the data type makes sense because an abstract class can't be implemented by a lambda expression.

---

[47] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/transaction/support/TransactionCallbackWithoutResult.html*

Understand execute(…) in TransactionTemplate

To better understand the `execute(…)` method of the `TransactionTemplate` class and the connection with the transaction manager, let's look at the slightly shortened implementation:[48]

```
public <T> T execute( TransactionCallback<T> action )
    throws TransactionException {
  …
  TransactionStatus status =
            this.transactionManager.getTransaction( this );
  // TransactionTemplate ..|> TransactionDefinition ^^^^
  T result;
  try {
    result = action.doInTransaction( status );
  }
  catch ( RuntimeException | Error ex ) {
    // Transactional code threw application exception -> rollback
    rollbackOnException( status, ex );
    throw ex;
  }
  catch ( Throwable ex ) {
    // Transactional code threw unexpected exception -> rollback
    rollbackOnException( status, ex );
    throw new UndeclaredThrowableException( ex,
      "TransactionCallback threw undeclared checked exception" );
  }
  this.transactionManager.commit( status );
  return result;
}
```

The approach of setting a `try-catch` block around `action.doInTransaction(status)` is something we programmed before. Now the `TransactionTemplate` class takes care of that.

Note: Log Transactions

By default, Spring Framework logs information about the beginning and end of transactions at the `DEBUG` level. However, since the default log level is set to `INFO`, these messages may not be visible to the user. In *application.properties*, for example, if you set the following:

```
logging.level.org.springframework.transaction=DEBUG
logging.level.org.springframework.orm.jpa=DEBUG
```

---

[48] *https://github.com/spring-projects/spring-framework/blob/5f0ee2e4dd60f8786c81ed56ea7637983145b7e8/spring-tx/src/main/java/org/springframework/transaction/support/TransactionTemplate.java#L130*

A log messages appear such as the following:

```
Getting transaction for […]
Completing transaction for […]
Initiating transaction rollback after commit exception
```

Task: Check Understanding of Transaction Continuation

Suppose we have a method called `one()` that executes a code block using `TransactionTemplate`. Additionally, we have a method called `many()` that performs the following operations:

1. Calls the `one()` method twice
2. Executes the method `one()` twice again within a `TransactionTemplate` code block

The code looks like this:

```
public void one() {
  transactionTemplate.execute( transactionStatus -> {
    Connection connection = DataSourceUtils.getConnection( dataSource );
    log.info( "{}", System.identityHashCode( connection ) );
    DataSourceUtils.releaseConnection( connection, dataSource );
    return null;
  } );
}

public void many() {
  one();
  one();

  transactionTemplate.execute( transactionStatus -> {
    Connection connection = DataSourceUtils.getConnection( dataSource );
    log.info( "{}", System.identityHashCode( connection ) );
    one();
    one();
    DataSourceUtils.releaseConnection( connection, dataSource );
    return null;
  } );
}
```

The question is: What do the log messages show?

Let’s take a look at one possible output:

```
… INFO 15604 --- [main] … : 12121212
… INFO 15604 --- [main] … : 89898989
… INFO 15604 --- [main] … : 1234567890
… INFO 15604 --- [main] … : 1234567890
… INFO 15604 --- [main] … : 1234567890
```

When the first two calls to the `one()` method are made within the `many()` method, they each occur in a separate database connection and transaction. However, when the two subsequent calls to the `one()` method are made within a `TransactionTemplate` block in the `many()` method, they continue the transaction in the existing connection. This is why the last three values are the same.

In this example, the method names `one()` and `many()` were deliberately chosen because it's a common pattern for an outer method to open the transaction and delegate subtasks to other methods that are executed within the existing transaction.

Note: Physical and Logical Transaction

When a program interacts with a JDBC `Connection` and directly manages the transaction, it's referred to as a *physical transaction*. The controls for this transaction are always carried out at the respective resource provider. To simplify management, a large physical transaction is often broken down into smaller *logical transactions*. Logical transactions are an abstraction and often used to simplify the management of complex database operations, as they allow developers to group multiple database operations into a single transactional unit.

5.11.6 @Transactional

Suppose there's a method called `deletePhotosById(…)` that performs all operations within a transactional context:

```
public void deletePhotosById( long... ids ) {
  transactionTemplate.execute( transactionStatus -> {
    …
  } );
}
```

Transactions are a *cross-cutting concern*, meaning that they are a common functionality required in various parts of a software system, but aren't directly related to the business logic. If we were to explicitly execute the transactional block in our code, as shown in the example, this would result in increased typing and code duplication. Fortunately, there is a better solution available for this implementation.

In the past, similar tasks were solved using proxies in Chapter 4. Proxies act as a wrapper around a method, allowing code to be executed before entering the core and after leaving the core. This is precisely what we need for transaction management, and Spring provides support for transactional proxies.

Programmed and Declarative Transaction Control

There are two ways to implement transaction controls. The first is through programming, using an API. Starting with the JDBC `Connection`, you can move to the `PlatformTransactionManager` and abstract the starting and ending of transactions via `TransactionTemplate`.

The second option is to work declaratively, without programming anything at all. This is done through the use of annotations, which provide a simple configuration. By setting the `@org.springframework.transaction.annotation.Transactional` annotation to a class or a method, Spring executes each method of the class or the annotated one in a transaction. The annotation must be evaluated via a proxy. If we use Spring Boot, an `@EnableTransactionManagement` is automatically applied via auto-configuration, enabling generation of the proxies that listen for whether something is `@Transactional`.

@Transactional

For a transactional block, we had previously used the `TransactionTemplate`:

```
public void deletePhotosById( long... ids ) {
  transactionTemplate.execute( transactionStatus -> {
    …
  } );
}
```

The annotation `@Transactional`[49] creates a proxy that does the same job as `TransactionTemplate`:

```
@Transactional
public void deletePhotosById( long... ids ) {
  …
}
```

As usual with proxies, the method must be `public`. If you call `deletePhotosById(…)` from outside, the proxy starts (or continues) the transaction and then calls our implementation `deletePhotosById(…)`. If there is a `RuntimeException` in our `@Transactional` method (at least this is the default), the transaction is rolled back; otherwise, it's committed.

Normally, methods are annotated with `@Transactional` in the business logic. The methods go to the backend, to the databases, via the corresponding repositories. We'll come back to to this topic of transactional processing later in Chapter 6, Section 6.8.5, when we discuss object-relational mapping.

Note

The annotation type `@Transactional` exists twice:

- **`@org.springframework.transaction.annotation.Transactional`**
  This is a Spring annotation that we should always use. It can be used to set all transaction-related information via annotation attributes, including the isolation level.

---

[49] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/transaction/annotation/Transactional.html*

- `@jakarta.transaction.Transactional`[50]
  This annotation comes from the Jakarta Transactions API (*Jakarta Transactions*). It provides only a few customizations via annotation attributes (`Transactional.TxType`, `rollbackOn`, `dontRollbackOn`).

In principle, Spring recognizes the Jakarta EE annotation, but the Spring annotation is preferable because it allows more settings.

5.12 Summary

In the previous chapter, the focus was on the basics of relational databases and how to interact with them at a low level of abstraction. However, in practical applications, working at such a low level isn't always necessary. Therefore, in the next chapter, the focus will shift to a more abstract level, where the goal is to map rows in the database to objects. This involves automating the process of retrieving data from the database and converting it into usable objects.

---

[50] *https://jakarta.ee/specifications/transactions/2.0/apidocs/jakarta/transaction/transactional*

Chapter 6

Jakarta Persistence with Spring

As Java developers, our programming world is centered around objects. Each object has three essential components: an identity, a state, and a behavior. These objects can interact with one another, forming relationships and connections. However, the true challenge arises when we try to merge the world of objects with the realm of database management systems—and that's what this chapter is about.

6.1 World of Objects and Databases

Of course, there is something else outside objects; in functional programming languages, for example, everything is a function. Database management systems generally don't store objects, but the data in documents, for example, or—and this is still the predominant way—in tables, which consist of rows and columns and contain the typed information in cells.

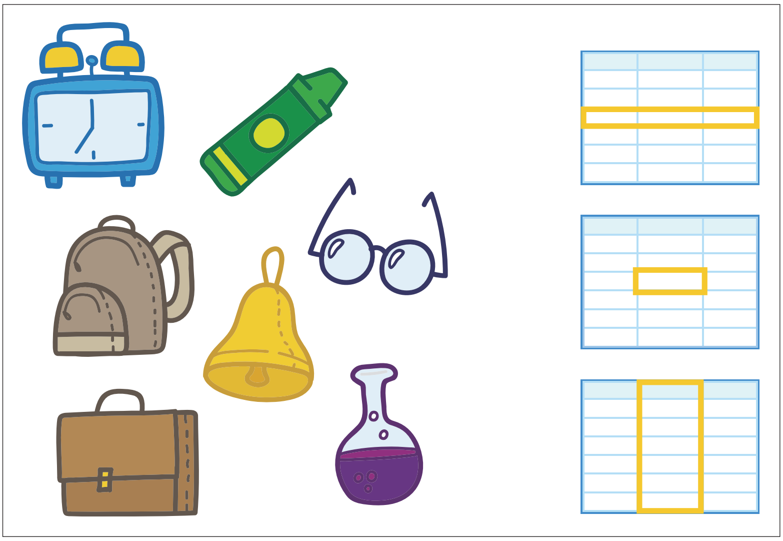


Figure 6.1 Arbitrary Objects with a Very Different Structure Than Tables

Objects and tables are far apart in their expressive power, as shown in Figure 6.1. Objects associate state with behavior, while relational databases focus on highly structured data.

6.1.1 Transient and Persistent

In the world of programming, objects can be classified as either *transient* or *persistent*. Transient objects only exist in the computer's main memory and are created using `new`. These objects have a finite life span, and if they are no longer referenced by the program, the garbage collector will clear them from memory. Additionally, if the Java virtual machine (JVM) is terminated due to an error or terminated manually, all transient objects and their values will be lost. On the other hand, databases operate differently. Any changes made to data within a transaction are written to the datastore, ensuring that the data isn't lost even in the event of a program failure.

It's possible to transfer object changes from the transient world to the persistent world through a magic mechanism. This process involves taking the transient objects from main memory and transforming them into persistent objects, which can be stored in databases. However, for this mechanism to work, the objects must be uniquely identifiable using a key or other identifier. By persisting objects in this way, their state can be saved and accessed even after the program has terminated, or the computer has been shut down.

6.1.2 Mapping Objects to Tables

It's possible to manually break down objects into their individual components and store them in a database. However, what programmers often prefer is an automated mapping process that can take complete objects (including associations and inheritance relationships) and convert all the relevant data into table format. Similarly, the mapping process can also retrieve data from tables and reconstruct it into objects, as illustrated in Figure 6.2. This automatic mapping process can save a significant amount of time and effort, as well as reduce the risk of errors that may arise from manual data manipulation.

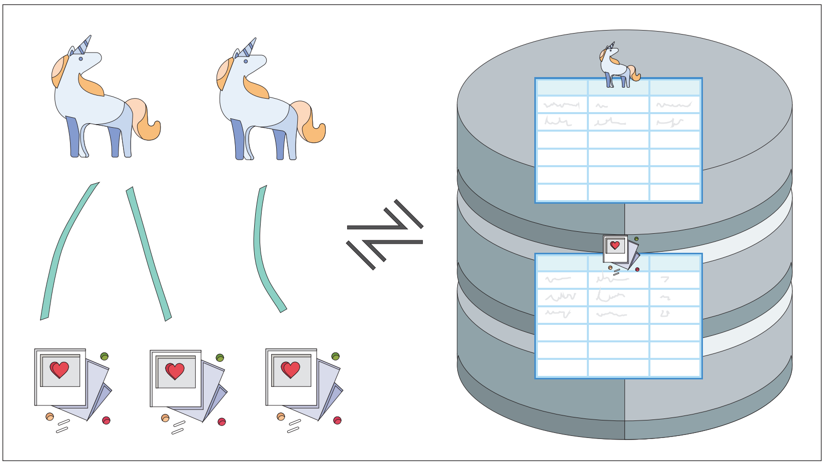


Figure 6.2 Desire: Object Graph Mapped into Tables

Creating SQL Statements

For objects to be written to or read from a database, SQL must be generated (see Figure 6.3).

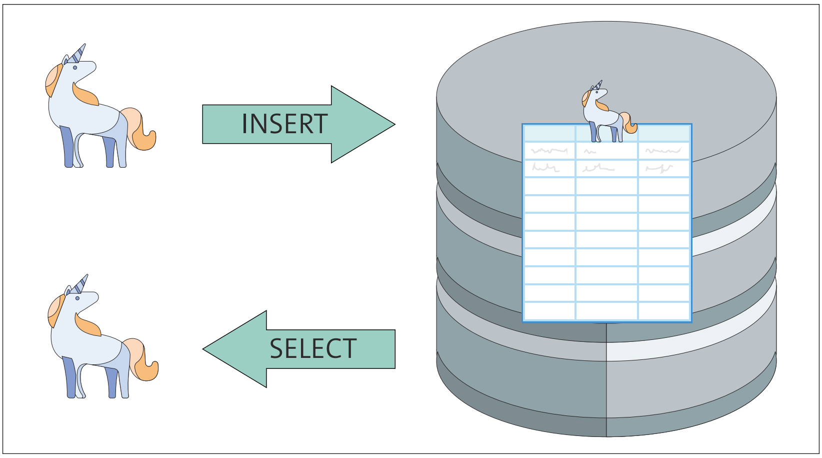


Figure 6.3 Magic Mechanism Generating SQL for the Mapping

When dealing with the automatic mapping of objects to databases, certain operations need to be performed to ensure that the data is properly stored and retrieved. For instance, when inserting objects into a database table, an `INSERT` statement is typically generated. Similarly, when reading data from the database, a `SELECT` statement is used. If a specific row in the database needs to be updated, an `UPDATE` statement is used to modify the relevant fields. It's important to note that not all database operations require a complete object to be created or modified. Sometimes, only a single property of a row in the database needs to be changed. In such cases, an `UPDATE` statement can be used to modify the specific property without altering the rest of the row. This approach can be useful for optimizing performance and minimizing unnecessary updates.

Object-Relational Mapping

The role of an *object–relational mapping* (*ORM* or *O/R mapper*) is to act as a mediator between the world of objects and the world of databases.[1] However, there are different perspectives on what functionalities an ORM must possess.

In previous chapters, we explored some form of ORM, such as through the use of `RowMapper`. However, relying solely on manual mapping can be time-consuming and inefficient. A fully functional ORM should automate the mapping process, requiring minimal code and configuration to read and write rows to the database.

---

[1] The term ORM is ambiguous because it can refer to either an O/R *mapper* or an O/R *mapping*.

Compared to the `JdbcTemplate`, which focuses primarily on rows and columns, an ORM places greater emphasis on the objects that are read or written. This shift in focus enables developers to work with objects more directly, rather than being bogged down in the technical details of database tables and queries.

Advantages of O/R Mappers

O/R mappers provide several advantages over manually mapping objects to relational databases:

- O/R mapper reduces the amount of code required, leading to increased productivity. Rather than spending time on manual mapping, the O/R mapper automatically maps tables to objects, saving developers time and effort.
- O/R mapper enhances maintainability by reducing the amount of code that needs to be changed in response to database changes, such as adding a column. With fewer lines of code to manage, it's easier to maintain the codebase.
- Complex SQL statements can be difficult to write and optimize. A good O/R mapper generates correct and optimized SQL statements, allowing developers to focus on higher-level tasks.
- The use of an O/R mapper promotes database independence by generating optimal SQL for any database. This eliminates the need to write database-specific code and reduces the risk of code becoming tightly coupled to a specific database.
- Performance is fine—if you know how to use the O/R mapper library properly. If the generated SQL statements aren't optimal, O/R mappers generally let you specify your SQL statements. In addition, there are intelligent caching mechanisms: if a program loads the same object multiple times, the O/R mapper recognizes this and can take the object from the existing session. In addition, lazy loading mechanisms can also be used: for example, if a 1:n association is read, the O/R mapper can load the dependent objects later when they are actually accessed. Multiple queries can often be combined by an O/R mapper, which reduces communication with the database.

Domain Model

When we develop object-oriented (OO) software, our focus should solely be on objects, as they are the fundamental building blocks that have their identity, state, and behavior. It's ideal to have a domain-specific model while modeling, which means that we concentrate on solving domain-specific problems. However, with ORM, there are two parties involved: the database administrators who think in terms of modeling data in tables, and the developers who think in terms of objects. Bridging these two worlds in ORM can be challenging. Although the data in the database and objects are related, mapping them correctly is the real challenge.

Impedance Mismatch

The primary issue with ORM is the *impedance mismatch* between the OO world and the table world. This mismatch arises due to the differing natures of these two worlds, as described here:

- In the OO world, every object has a unique identity, whereas on the database side, rows are identified by keys. Thus, when mapping objects to the database, a key must be assigned to each object, which wasn't required earlier in the Java side.
- On the Java side, object references make it easy to navigate to other objects, for example, using expressions such as `customer.getAddress().getCity().getName()`. However, transferring such references to the database side can be complex as it requires key relationships to be established. On the database side, these relationships are resolved using a `JOIN`, which can be expensive in terms of performance. In contrast, on the Java side, navigating in memory is almost free and doesn't have any performance overhead.
- In Java, associations with 1:1 or 1:n relationships are always mapped consistently. However, on the database side, there are multiple ways to map these relationships. For example, a separate join table could be used, or a row could have a join column. These techniques are completely distinct from each other. In Java, the use of join objects to combine different objects isn't required.
- Java uses inheritance to represent type relationships between classes. For example, we might have a class called `Container` and subclasses called `Bottle` and `Glass`. These subclasses can be seen as types of the `Container` class. However, representing this relationship in a database can be challenging. There are several techniques that can be used, each with its drawbacks. Despite these challenges, on the Java side, we would typically use inheritance to model "is a type of" relationships.

To summarize, mapping objects to tables and databases presents several challenges. Neither the OO side nor the database schema can enforce a relationship on the other side. This means that starting with a domain model in Java may not result in a good database model, and starting on the database side may not produce a good domain model for the OO world. Therefore, it's up to database admins and OO developers to find the best modeling in their respective worlds. An O/R mapper must then bridge the gap between these two worlds.

Disadvantages of O/R Mappers

O/R mappers are useful, but they also bring some challenges:

- O/R mappers require a greater learning curve. Unlike imperative programming with the `JdbcTemplate`, ORM is more declarative and may seem "magical."
- You must trust that the O/R mapper generates good SQL that you would have written yourself. Understanding the generated SQL is essential for identifying potential

performance issues. If you don't understand the O/R mapper, it may generate expensive queries.

- It may be easier to write SQL directly instead of using the O/R mapper. Although the O/R mapper can solve any problem, the solution may be difficult to understand and maintain.
- If the Java domain model is vastly different from the database model, mapping the models can be a significant challenge.
- If objects aren't necessary, such as when changing the timestamp of photos due to time zone adjustments, loading each photo individually as an object may not be efficient. In this case, it may be faster to modify the photos directly without using objects.

Overall, O/R mappers are just one tool among many. While they have weaknesses, they can make everyday life easier. It's important to determine where O/R mappers are most useful, how to address their weaknesses, and when they may hinder progress. Remember, there is no obligation to use O/R mappers; when necessary, Java Database Connectivity (JDBC) drivers and SQL can be used instead.

6.1.3 Java Libraries for O/R Mapping

An O/R mapper is a concept, not a product. On the Java side, there are numerous libraries that perform ORM. These solutions vary in complexity and power, and they operate at quite different levels of abstraction. Here's a selection:

- Apache Cayenne (*https://cayenne.apache.org*)
- Doma (*https://github.com/domaframework/doma*)
- Ebean ORM (*https://ebean.io*)
- **EclipseLink** (*www.eclipse.org/eclipselink*)
- **Hibernate ORM** (*https://hibernate.org/orm*)
- jOOQ (*www.jooq.org*)
- MyBatis (*https://mybatis.org*)
- ObjectiveSql (*https://github.com/braisdom/ObjectiveSql*)
- OrmLite (*https://ormlite.com*)
- Persism (*https://sproket.github.io/Persism/*)
- Querydsl (*https://querydsl.com*)
- Reladomo (*https://github.com/goldmansachs/reladomo*)
- requery (*https://github.com/requery/requery*)
- Spring Data JDBC (*https://spring.io/projects/spring-data-jdbc*)
- Sql2o (*www.sql2o.org*)

The list presents several O/R mappers, but many of them are created by individuals or small teams. It's crucial to have a supported standard that large IT companies can adopt.

Out of all the solutions listed, EclipseLink and Hibernate ORM are emphasized for their adherence to a standard that was originally defined by Sun Microsystems and is currently maintained by Oracle for ORM.

6.2 Jakarta Persistence

About 20 years ago, Java Enterprise Edition (J2EE) application servers had a feature called *container-managed persistence* (*CMP*) for storage. However, it was complex to use and tightly coupled to the application server.

To solve this problem, in 2006, Sun created a new standard, the *Java Persistence API* in a rewrite of *Enterprise JavaBeans 3.0* (JSR 220, *https://jcp.org/en/jsr/detail?id=220*) as part of the Java EE specifications. JPA 1.0 is an elementary component of the Enterprise JavaBeans standard. The Java Persistence API replaced the EJB 2.0 CMP.

In 2009, the Java Persistence API broke away from the Enterprise JavaBeans standard and became *Java Persistence 2.0* (JSR 317, *https://jcp.org/en/jsr/detail?id=317*). This made it possible to use the Java Persistence API outside an enterprise application server, for example, in regular Java SE applications. Oracle acquired Sun Microsystems in the same year.

Java Persistence API 2.0 was followed by Java Persistence API 2.1 and later Java Persistence API 2.2, both of which are described together in JSR 338 (*https://jcp.org/en/jsr/detail?id=338*).

During that time, Oracle interest in Java EE and handed over the standards to the Eclipse Foundation. In this transition, the Java Persistence API had to be renamed and is now called *Jakarta Persistence* (*https://jakarta.ee/specifications/persistence/*). In 2019, Java Persistence 2.2 became Jakarta Persistence 2.2, which is the standard for Spring Boot 2 applications.

One consequence of the breakaway from Sun was that the package name `javax` was abandoned, resulting in the base package `jakarta`. Jakarta Persistence 2.2 was followed in 2020 by Jakarta *Persistence 3* (*https://jakarta.ee/specifications/persistence/3.0*) as part of Jakarta Enterprise Edition version 9 with the new package name. Thus, the former `javax.persistence` namespace ultimately became `jakarta.persistence`. Jakarta Persistence 3.0 is also the basis for Spring Framework 6 and Spring Boot 3.

Jakarta Persistence 3.1 (*https://jakarta.ee/specifications/persistence/3.1/*) was released in 2022 and is the release for Jakarta EE 10. The minimum requirement for Spring Data JPA 3.0 is Jakarta Persistence 3.0, but Spring Boot 3.0 already includes an implementation for Jakarta Persistence 3.1 via auto-configuration, which is why the links in this book point to version 3.1.

6.2.1 Persistence Provider

A standard is in itself a large document with many rules. The Jakarta Persistence specification isn't small at 600 pages, but it's manageable and easy to read compared to the SQL standard.[2]

Because you can't execute a specification, an implementation is needed, which is called a *persistence provider*. Two of the most important implementations are Hibernate ORM and EclipseLink JPA:

- **Hibernate ORM**
  This is a solution from Red Hat and is licensed under the *Lesser General Public License (LGPL)*. Hibernate ORM is very widely used. Among other things, it's the default persistence provider that Spring Boot includes by default. Red Hat (i.e., IBM) also uses Hibernate ORM for its own application servers, *Wild Fly* and *JBoss EAP*. The Hibernate homepage lists other products with the name "Hibernate," for example, *Hibernate Validator*, *Hibernate Search*, and *Hibernate Tools*. However, we always want to use the term Hibernate to mean the O/R mapper. Strictly speaking, the O/R mapper is called *Hibernate ORM*, but simplified we just say Hibernate.
- **EclipseLink JPA (EclipseLink for short)**
  This is a second important implementation and the reference implementation of the specification. It's available under the free Eclipse license. In addition, EclipseLink has known users, for example, application server *GlassFish* (reference implementation of the Jakarta Enterprise specification) and *Payara* (based on GlassFish, only there are also commercial licenses). From EclipseLink, there is an extension called *TopLink*, which is a product of Oracle. TopLink is used as a Java Persistence API implementation in the application server *Oracle WebLogic Server*.

In addition to Hibernate ORM and EclipseLink, *Apache OpenJPA* (*https://openjpa.apache.org*) is another option for persistence providers. However, data storage is a critical and essential aspect of software development, and it's not ideal to experiment with incomplete or unreliable solutions. We must ensure that data isn't lost or improperly written. Therefore, we'll rely on Hibernate ORM, which has been in use in production environments for more than two decades and is thoroughly tested.

Note

Spring Boot 3 references Hibernate ORM version 6.1, which in turn implements Jakarta Persistence Specification 3.1. Spring Boot 3.1 upgrades to Hibernate ORM 6.2.

---

[2] For comparison, *ISO/IEC 9075-2:2016 Part 2: Foundation (SQL/Foundation)* has more than 1,300 pages and is only part of the 10 SQL publications. (In total, we're dealing with around 4,000 pages for SQL.)

6.2.2 Jakarta Persistence Provider and JDBC

A Jakarta Persistence provider uses the JDBC driver internally to talk to the database. An O/R mapper builds SQL in the background and sends it via the JDBC API. The JDBC API is again just an API that is implemented by drivers (see Figure 6.4).

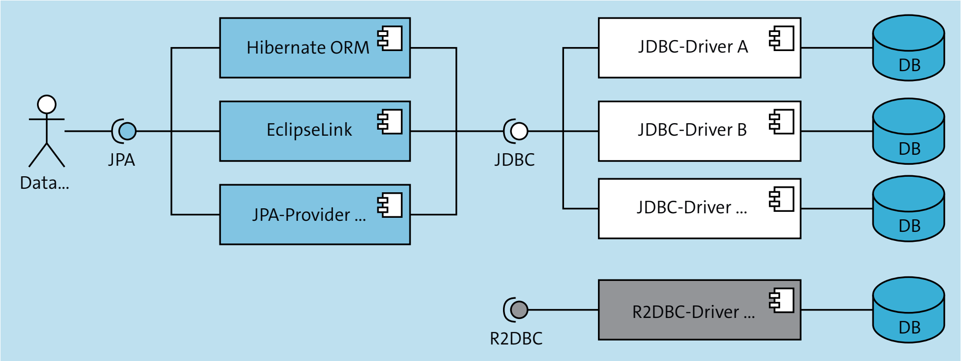


Figure 6.4 The Flow: Client Uses JPA, Implementation Accesses JDBC API, and the JDBC API Accesses the Database

In addition to the traditional JDBC API, there is also the reactive Reactive Relational Database Connectivity (R2DBC) API for asynchronous access to databases. However, there is currently no direct integration between Jakarta Persistence API and R2DBC due to the synchronous nature of the Jakarta Persistence API and the blocking nature of JDBC. While Hibernate offers a reactive implementation (*https://hibernate.org/reactive*), it's not part of the Jakarta Persistence standard. As of now, there are no plans for a reactive Jakarta Persistence standard, but this could change in the future.

6.2.3 Jakarta Persistence Coverage

The *Jakarta Persistence specification* is composed of three main sections that are essential for storing and retrieving entity beans in a database. An *entity bean* is a Java object that has an identity and can be saved in a database.

- **EntityManager API**
  As the first section of the Jakarta Persistence API, the EntityManager API is responsible for making entities accessible on the Java side. The EntityManager API is similar to the JDBC API, but instead of operating at the row level, it operates on the entity beans. The EntityManager API provides methods such as `persist(…)` or `find(…)` to manage entities. In addition to the EntityManager API, the Query API and the Criteria API are also available. These APIs can further parameterize the queries.
- **Jakarta Persistence Query Language (JPQL)**
  This makes up the second section of the Jakarta Persistence specification. JPQL is a query language that can be used to select, delete, or update rows in the database.

JPQL works with entities, not tables, and is similar to OO SQL. JPQL statements are converted to SQL statements and executed by the database.

- **Mapping descriptions**
  The third section deals with mapping descriptions. Metadata is described declaratively through annotations or XML descriptors. The metadata describes how the entity bean properties are mapped to database columns. The metadata is essential for the entity bean to be stored in the database. Annotations are the most commonly used way to describe metadata, but it's also possible to use XML descriptors, such as the *orm.xml* file. By using external files, the source code can remain free from any technical dependencies, such as ORM. However, for simplicity, annotations are used in the code in this case.

Next, we'll include the persistence provider *Hibernate* in a Spring Boot project.

6.3 Spring Data JPA

In the practical realm, we can delve into a Spring Boot project that employs the Jakarta Persistence API. Our objective is to examine the various components of the API and showcase their usage in fetching and storing data from a database.

6.3.1 Include Spring Boot Starter Data JPA

We're going to fall back on a project called *Spring Boot Starter Data JPA*. There is a starter for Spring Boot in the following listing.

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
```

Listing 6.1 pom.xml Extension

From our previous project, we can change `spring-boot-starter-jdbc` to `spring-boot-starter-data-jpa` because Spring Boot Starter Data JPA includes Spring Boot Starter JDBC.

The Spring Boot Starter Data JPA references three things at its core:

- Spring Data JPA
- Hibernate ORM
- Spring Boot Starter JDBC

The Spring Boot Starter JDBC is included because that takes care of the preconfigured `DataSource` and also the connection pool, for example.

6.3.2 Configurations

Further configurations aren't really necessary, but we want to make two small settings.

Switch Off Auto-Generation

One important consideration is that a Jakarta Persistence provider can modify databases to match the structure of the entities. While it may be helpful to automatically create tables, it can be alarming if the provider deletes a table and erases all of its data just to make a minor change to a column.

There are two ways to stop this habit in the Jakarta persistence provider. The first is to set configuration property `spring.jpa.generate-ddl` to `false`. DDL here stands for *Data Definition Language*, that is, the SQL statements `CREATE TABLE`, `DROP TABLE`, and `ALTER TABLE`. Spring transfers the configuration property to the corresponding persistence provider, so this switch works with Hibernate ORM and also with EclipseLink JPA.

There is another way, and that is the concrete Jakarta Persistence providers themselves have their configuration properties. For Hibernate, this is `spring.jpa.hibernate.ddl-auto`; one assignment is `validate`.

```
spring.jpa.hibernate.ddl-auto=validate
```

Listing 6.2 application.properties Extension

The `validate` attribute ensures that Hibernate checks if the entities in the application match the database schema during startup. If there are any discrepancies or mapping issues, Hibernate will log an appropriate message.

Other values are `none`, `validate`, `update`, `create`, and `create-drop`.[3] For a production system, `validate` shouldn't be necessary, as the entity beans should match the database schema, and the test can be turned off to gain a bit of performance at startup time. `create-drop` can be interesting for test cases, so that Hibernate builds the tables at the beginning. `update` means that only tables are modified, but not deleted.

Log-Generated SQL

An O/R mapper generates SQL, and it's important to understand the generated SQL. Hibernate allows a configuration by which it logs all generated SQL statements. This also allows performance problems to be detected at an early stage. Specifically, the *SELECT N+1 update problem* can be recognized quickly because then it permanently rattles in the console.

```
logging.level.org.hibernate.SQL=DEBUG
logging.level.org.hibernate.orm.jdbc.bind=TRACE
```

---

[3] *https://docs.spring.io/spring-boot/docs/current/reference/html/howto.html#howto.data-initialization.using-hibernate*

```
spring.jpa.properties.hibernate.format_sql=true
spring.jpa.properties.hibernate.highlight_sql=true
```

Listing 6.3 application.properties Extension

Hibernate automatically logs output at the `DEBUG` level, so `org.hibernate.SQL` is downgraded so that the messages are visible. If `logging.level.org.hibernate.orm.jdbc.bind`[4] is set to `TRACE`, this will output the call parameters of the prepared statements. Of course, we must be careful because sensitive data could end up in the log.

There are alternative approaches to formatting the output of Hibernate. By default, Hibernate presents the SQL statements on a single line, which is suitable for log preservation but may not be easily legible. This is why setting `format_sql=true` enables the formatting of output in multiple lines with proper indentation. Another intriguing feature of Hibernate is `highlight_sql`, which, when set to `true`, displays a colorful output of keywords on a modern console window.

Hibernate provides the ability to generate statistics that can be viewed for analysis purposes. To enable this feature, we need to set the property in the following listing.

```
spring.jpa.properties.hibernate.generate_statistics=true
```

Listing 6.4 application.properties

These statistics reveal useful information such as the number of connections opened or the execution time of instructions.

If production systems are to show maximum performance, the properties could be occupied as shown in the following configuration.

```
spring.jpa.properties.hibernate.generate_statistics=false
logging.level.org.hibernate=ERROR
```

Listing 6.5 application.properties

Note

In principle, configuration property `spring.jpa.show-sql` can also be set to `true`, but this results in console output. However, we want the data in the logger, which can be directed to arbitrary channels.

P6Spy can also log. The advantage is that P6Spy wraps itself around every `DataSource` and can log everything, regardless of whether Hibernate or another O/R mapper is used. However, because Hibernate formats nicely, we'll stick with the configurations we've made.

---

[4] In Hibernate 5, it was `logging.level.org.hibernate.type.descriptor.sql.BasicBinder`. If you migrate applications from Spring Boot 2 to Spring Boot 3, you have to adapt this.

6.4 Jakarta Persistence Entity Bean

Entity beans, also known as persistent objects, have a key property that distinguishes them from regular Java objects: a primary key uniquely identifies them. Mapping data from tables to the states of entity beans can pose a challenge due to the impedance mismatch problem between the object and table worlds, as we discussed in Section 6.1.2.

To address this problem, two approaches are typically taken: the code-first approach or the database-first approach.

- *Code-first approach*
  In software development, we start with the entity beans. The database is determined based on the OOP models. This approach is unusual in practice, as there are usually predefined databases with tables and schemas. In addition, performance on the database side would not necessarily be optimal.
- *Database-first approach*
  We start with the database and write (or automatically generate) the entity beans; the database modeling comes first. The disadvantage could be that the entity beans don't feel like full-fledged objects, but they are just blunt data containers; this antipattern is also called *anemic domain model* (*www.martinfowler.com/bliki/AnemicDomainModel.html*).

Neither approach is optimal. In practice, there are mixed forms that work well—the database schema is always in the foreground:

- The entity beans try as best as possible to be "good objects," and all the Jakarta Persistence possibilities for mapping the tables to objects are exhausted.
- Without even trying to make the entity beans "good" objects, this method takes an almost automatic 1:1 mapping of the tables to the entity beans. However, these entity beans remain internally in the infrastructure layer and are mapped to "real domain objects" with the perfect data types—then no more technological annotations or details come through. If you want to achieve the best possible domain-driven design, you'll go this route, but the effort is high, and the practical benefit is rather low.

With our database schema, this is simple: we can transfer the tables directly to entity beans. There is no case where the data of an entity bean is distributed to multiple tables or a table is distributed to multiple entity beans.

6.4.1 Develop an Entity Bean Class

Persistent objects in Java are implemented using entity beans, with each entity bean being a basic Java object. To represent an object, a class is required to act as a container for its states. In this case, let's create a new class called `Profile` that can later hold the details of the `Profile` table:

```
class Profile {
}
```

You don't have to implement interfaces or extend a specific superclass. The class names usually don't have special prefixes or suffixes; we can put the type in a package `entity` and separate it from other types. That the class is named like the table is common, as long as the table follows the Java naming conventions.

Data Type Mapping

For the Jakarta Persistence provider to map rows from the `Profile` table to an entity bean instance, the entity bean must have the ability to hold the data. It's important to determine which data from the table should be held by the entity beans and which Java data types are appropriate.

For this discussion, let's focus on the `Profile` table. Because we're using the database-first approach, we need to examine all the columns that we want to transfer to the entity bean. In our case, we'll transfer all the columns, as listed in Table 6.1.

Column Name	SQL Type	Java Type
`id`	`BIGINT`	`Long`, `long`
`nickname`	`VARCHAR(32)`	`String`
`birthdate`	`DATE`	`java.time.LocalDate`
`manelength`	`SMALLINT NOT NULL`	`short`
`gender`	`TINYINT NOT NULL`	`byte (TINYINT` 0-255)
`attracted_to_gender`	`TINYINT`	`Byte`
`description`	`VARCHAR(2048)`	`String`
`lastseen`	`TIMESTAMP`	`java.time.LocalDateTime`

Table 6.1 Data Types on the Java and Database Side

The transfer of the types happens via two dimensions: How big are the data types, and can they be `NULL`? If they can be `NULL` (called *nullable types*), we must use the wrapper types (`Long`, `Short`, `Byte`) instead of the primitive data types because references can be `null`.

In some cases, there are several options:

- With `id`, there is the peculiarity that `Long` or `long` would fit depending on the use case. A wrapper `Long` can be `null`, a primitive `long` of course can't, and `0` isn't equal to `null`. In our program, `Long` will be necessary because there is a short time when the ID isn't assigned, and that is when a `Profile` is built with `new`, and the database saves the row and writes back an ID. If we were to just read from the database, then we would always get finished and built `Profile` objects and the variable would always be assigned and never `null`. In this case, `long` would be fine. In addition, `long` would be fine if we assigned IDs ourselves, but we don't do that.
- `birthdate` is a date, and besides `LocalDate`, the older data types such as `java.util.Date` or `java.sql.Date` would also be conceivable. Fortunately, Jakarta Persistence has been able to handle the temporal data types such as `LocalDate`, `LocalTime`, and `LocalDateTime` without any problems for some time now.
- The data type for `gender` is more challenging. On the database side, the column is defined as `TINYINT`, which corresponds to a `byte`/`Byte` on the Java side. However, there is an issue with `TINYINT` being an unsigned byte on the database side, which means it can range from 0 to 255. On the other hand, Java doesn't have unsigned data types, and its `byte` values range from -128 to +127. Despite this disparity, because `gender` only involves small numbers, it still falls within the acceptable range.
- `lastseen` is a `TIMESTAMP`, and `LocalDateTime` is a good choice. In principle, the data types `java.util.Date` and `java.sql.Timestamp` are also possible.

Our data types are thus fixed. We can set instance variables with the selected data types in the `Profile` class:

```
public class Profile {
  Long id;
  String nickname;
  LocalDate birthdate;
  short manelength;
  byte gender;
  Byte attractedToGender;
  String description;
  LocalDateTime lastseen;
}
```

For some O/R mappers, this is sufficient. However, Jakarta Persistence has further requirements.

O/R Metadata

With Jakarta Persistence, more O/R metadata is needed because the instance variables alone aren't enough. The metadata determines, for example, that the class is an entity bean or what the name of the join column is.

The Jakarta Persistence specification provides two ways to specify the metadata:

- **Annotations**
  This is the usual way.
- **XML file**
  An XML file in the *META-INF* directory with the file name *orm.xml* contains all descriptions.

Both ways are equally expressive. The advantage (and disadvantage) of the annotations is the direct connection to the code, but the settings are also directly exposed this way. The Jakarta Persistence specification documents both variants.

We'll later set metadata via annotations, and these can be roughly divided into two groups:

- **Annotations for logical mapping**
  These annotations describe the object model, associations, and so on.
- **Annotations for the physical mapping**
  These annotations describe the database schema such as table name, index, and so on.

O/R mappers require the metadata for correct mapping. The entity bean class carries a logical annotation:

```
@Entity
public class Profile {
  …
}
```

A Jakarta Persistence entity class is always annotated with `@Entity`; the annotation is allowed on concrete or abstract classes, but not on interfaces.

The Jakarta Persistence provider must take the data from the JavaBean and write it back. In doing so, it can fall back on the instance variables or on setters/getters:

```
@Entity
@Access( AccessType.FIELD )
public class Profile {
  …
}
```

By using the `@Access(AccessType.FIELD)` annotation, the Jakarta Persistence provider is informed that the data is stored within the object's variables instead of being accessed through setter/getter methods. Further details regarding this matter can be found in Section 6.10.4.

Let's add the instance variables:

```
@Entity
@Access( AccessType.FIELD )
public class Profile {
  @Id                                                ❶
  @GeneratedValue(strategy = GenerationType.IDENTITY) ❷
  private Long id;

  private String nickname;
  private LocalDate birthdate;
  private short manelength;
  private byte gender;

  @Column(name = "attracted_to_gender")              ❸
  private Byte attractedToGender;

  private String description;
  private LocalDateTime lastseen;
}
```

❶ The primary key is called *Id-Property* and is annotated with the Jakarta annotation `@Id`.[5] This is a logical annotation.

❷ Auto-generated keys get `@GeneratedValue`[6] as well as a strategy. We'll look at this in Section 6.10.9 in more detail.

❸ In many places, metadata for the physical mapping can be omitted because the Jakarta Persistence provider performs an *implicit mapping*. For example, if an entity class is called `Profile`, the Jakarta Persistence provider assumes that the table is also called `Profile`. It's the same with persistent attributes: if a persistent attribute is called `id`, the Jakarta Persistence provider assumes that the column is called the same. In this context, it's convenient that SQL is case-insensitive. If you want the name of the column to be different from the persistent attribute name, you use `@Column`. In the code, `attractedToGender` is mapped to the `attracted_to_gender` column. Spring Boot has a setting that does this automatically, but you'll learn more on why this is so later in Section 6.10.6.

Each entity bean class must have at least one `public` or `protected` parameterless constructor; the persistence provider will use reflection to automatically build the object later.

```
@Entity
@Access( AccessType.FIELD )
public class Profile {
  …
  protected Profile() { }
```

---

[5] *https://jakarta.ee/specifications/persistence/3.1/apidocs/jakarta.persistence/jakarta/persistence/id* not `org.springframework.data.annotation.Id`.

[6] *https://jakarta.ee/specifications/persistence/3.1/apidocs/jakarta.persistence/jakarta/persistence/generatedvalue*

```
  public Profile( String nickname, LocalDate birthdate,
                  int manelength, int gender,
                  Integer attractedToGender, String description,
                  LocalDateTime lastseen ) {
    setNickname( nickname );
    setBirthdate( birthdate );
    setManelength( manelength );
    setGender( gender );
    setAttractedToGender( attractedToGender );
    setDescription( description );
    setLastseen( lastseen );
  }
}
```

Parameterized constructors are handy when our code builds new profiles, so the ID is also missing because we don't determine it.

The constructor calls setters and initializes the object. In addition, the client can later change the states with the setters and read the states with getters. The implementation of the setters/getters is as usual, and the following are as expected:

- `String getNickname()`
- `void setNickname( String nickname )`
- `LocalDate getBirthdate()`
- `void setBirthdate( LocalDate birthdate )`
- `String getDescription()`
- `void setDescription( String description )`
- `LocalDateTime getLastseen()`
- `void setLastseen( LocalDateTime lastseen )`
- `Long getId()`

For ID, there is only a getter, but no setter.

For three persistent attributes, the setters/getters perform a type conversion:

```
  public int getManelength() {
    return manelength;
  }
  public void setManelength( int manelength ) {
    this.manelength = (short) manelength;
  }

  public int getGender() {
    return gender;
  }
  public void setGender( int gender ) {
    this.gender = (byte) gender;
  }

  public @Nullable Integer getAttractedToGender() {
    return attractedToGender == null ? null : attractedToGender.intValue();
  }
  public void setAttractedToGender( @Nullable Integer attractedToGender ) {
    this.attractedToGender = attractedToGender == null ?
                                 null : attractedToGender.byteValue();
  }
```

This is because using the data types `byte` and `Byte` in Java can be somewhat cumbersome, whereas `int` and `Integer` are more user-friendly. Converting between data types can be helpful, and it could potentially enable the use of `Optional*` for nullable types. Instead of annotating with `@Nullable Integer`, it would be just as appropriate to use the API's `OptionalInt`.

We're almost done with the entity bean class. In the last step, we add methods from `java.lang.Object`:

```
  @Override public boolean equals( Object o ) {
    return o instanceof Profile profile
           && Objects.equals( nickname, profile.nickname );
  }

  @Override public int hashCode() {
    return Objects.hashCode( nickname );
  }

  @Override public String toString() {
    return "%s[id=%d]".formatted( getClass().getSimpleName(), getId() );
  }
}
```

In the implementation of methods `equals(…)` and `hashCode()`, only the unique nickname or *business key* is used. The `Objects` methods can be helpful in cases where `equals(…)` or `hashCode()` uses the nickname early, even before initialization, and the instance variable is `null`.

When it comes to method `toString()`, it only accesses the ID to avoid unnecessary reloading and cycles. Cycles can occur when, for example, a profile has a photo and includes the `toString` representation, but the photo references back to the profile, creating an endless loop. To avoid such situations, only the ID is used in the `toString()` method.

Tool Support

Development environments such as IntelliJ help map tables to objects. For example, IntelliJ can automatically generate the entities from the database tables. However, this is often only a start, and manual work is required.

For IntelliJ, there is also a plugin called *JPA Buddy* (*www.jpa-buddy.com*) with useful features that aren't yet integrated in IntelliJ. JPA Buddy helps in the following situations:

- During database migration
- When generating repositories
- When creating repository methods via the GUI

JPA Buddy also doesn't require an Ultimate edition.

In this section, we've introduced the first entity bean, with additional beans for photos and unicorns to follow.

Moving forward, we'll focus on how to efficiently query, store, update, and delete these entities. To assist with these tasks, we'll use a specialized data type known as the `EntityManager`.

6.5 Jakarta Persistence API

In a Jakarta Persistence application, all types are sourced from the `jakarta.persistence` package. This API is comprehensive, and with Jakarta Persistence 3.1, it includes 16 interfaces (originally, there were only 4: `EntityManager`, `EntityManagerFactory`, `EntityTransaction`, and `Query`), a `Persistence` class (relevant only for Java SE applications), 19 enumerations, 11 exception classes, and 91 annotations for entity beans.

The meaning of the annotations can be easily understood by examining the types; specifications are set declaratively and not programmed via an API.

The `EntityManager` interface is perhaps the most important among all and it offers about 50 methods. All interfaces are implemented by the respective persistence provider, comparable to how a JDBC driver implements the JDBC API.

6.5.1 Getting the EntityManager from Spring

After we've programmed the entity bean, we can turn our attention to the API of the `EntityManager`. This allows us to load, save, and delete the entity bean. We want to read with the `EntityManager`[7] first, then write later. Transactions are necessary for writing.

---

[7] *https://jakarta.ee/specifications/persistence/3.1/apidocs/jakarta.persistence/jakarta/persistence/entitymanager*

Inject EntityManager

The `EntityManager` represents an open session with the database. Spring Boot builds an `EntityManager` via auto-configuration, which we can inject directly:

```
@Autowired EntityManager em;
```

An alternative possibility offers the annotation `@PersistenceContext`.[8] This is an annotation from the Jakarta Persistence.

Inject the EntityManager and Test If It Exists

For access via the `EntityManager` to work, several things must fit together: The database must be running, the connection data must be correct so that a connection can be established, the entity bean class must be found, and the entity bean metadata must match the database schema.

To check if all of this is in order, the demo in the following listing might help.

```
package com.tutego.date4u.interfaces.shell;

import org.slf4j.LoggerFactory;
import org.springframework.shell.standard.*;
import jakarta.persistence.EntityManager;
import static java.util.Objects.isNull;

@ShellComponent
public class EntityManagerCommands {

  private final EntityManager em;

  public EntityManagerCommands( EntityManager em ) { this.em = em; }

  @ShellMethod( "Display profile" )
  public String findprofile( long id ) {
    LoggerFactory.getLogger( getClass() ).info( "{}", em );
    return isNull( em ) ? "null" : em.getClass().getName();
  }
}
```

Listing 6.6 EntityManagerCommands.java

If we call `findprofile` from the shell—the ID is ignored by the code for now—a `Class` object identifier should appear, but no exception or `null` should show up on the console.

---

[8] *https://jakarta.ee/specifications/persistence/3.1/apidocs/jakarta.persistence/jakarta/persistence/persistencecontext*

Note: Working with the EntityManager API?

Spring applications can use the Jakarta Persistence API in two ways. First, the EntityManager API can be used directly, and, second, there is the *Spring Data project*, particularly *Spring Data JPA*, which works in the background with the `EntityManager`. While working with Spring Data JPA eventually, we won't have direct contact with the `EntityManager`. However, it's essential to learn the basics of the EntityManager API as it helps us understand the persistence context and how entity beans are managed and monitored in memory. Even if Spring Data JPA makes the EntityManager API obsolete, the familiar annotations and JPQL as a query language remain relevant.

Now let's look at some core `EntityManager` methods. You'll find the Javadoc link at *https://jakarta.ee/specifications/persistence*.

6.5.2 Search an Entity Bean by Its Key: find(…)

Method `find(…)` in `EntityManager` searches for an entity bean using a given key. Its declaration is as follows:

```
<T> T find(Class<T> entityClass, Object primaryKey)
```

The first parameter is a type token, which is precisely the type of entity that is to be loaded. The second parameter is the primary key; this is of type `Object` because the key can be of different types. For example, load the profile with ID 1:

```
Profile fillmoreFat = em.find( Profile.class, 1L );
```

In our case, the primary key is `1L`, which leads to `Long.valueOf(1)`, a wrapper object. The keys are always objects; this is where autoboxing is helpful. If the data types aren't correct, for example, because the type token doesn't belong to any entity bean or the key type is wrong, this leads to an `IllegalArgumentException` at runtime, that is, an unchecked exception.

If the `find(…)` method finds nothing, the return is `null`. This is rather uncommon in modern API design because you would use `Optional` today; the EntityManager API is much older than Java 8. If you prefer to work with `Optional` and like the cascading with `map(…)`, you can work with `Option.ofNullable(…)`:

```
Optional<Profile> maybeProfile =
     Optional.ofNullable( em.find(Profile.class, 1L) );
```

The `Optional` either contains a reference to the loaded entity bean or is `Optional.empty()` if there was no profile under the ID.

Task: Find and Output Profiles with ID

We've described in Section 6.5.1 under the "Injecting EntityManager" subsection how we started with the shell method `findprofile`:

```
@ShellMethod( "Display profile" )
public String findprofile( long id ) {
  LoggerFactory.getLogger( getClass() ).info( "{}", em );
  return isNull( em ) ? "null" : em.getClass().getName();
}
```

The `findprofile` command should evaluate the parameter so that a profile with the given ID is loaded and displayed.

Proposed solution:

```
@ShellMethod( "Display profile" )
public String findprofile( long id ) {
  return Optional.ofNullable( em.find( Profile.class, id ) )
      .map( profile->profile.getNickname()+", "+profile.getManelength() )
      .orElse( "Unknown profile for ID " + id );
}
```

The proposed solution uses the previously discussed method of using `Optional.ofNullable(…)` to convert the reference to an `Optional`. The `map(…)` method transforms the `Optional<Profile>` to an `Optional<String>` with the nickname and mane length. Because our shell method must always return a string, `orElse(…)` ensures that the string `"Unknown profile for ID " + id` appears at the end in case the `find(…)` method failed to load the entity and returned `null`, resulting in `Optional.empty()`.

If we load the profile with ID 1, the following SQL[9] is generated and displayed:

```
select
    p1_0.id,
    p1_0.attracted_to_gender,
    p1_0.birthdate,
    p1_0.description,
    p1_0.gender,
    p1_0.lastseen,
    p1_0.manelength,
    p1_0.nickname
from
    profile p1_0
where
    p1_0.id=?
```

---

[9] Since Hibernate 6, the generated SQL looks different. In older versions, Hibernate worked by column name, so the generated SQL would look like: `profile0_.id as id1_0_0_`. The current versions of Hibernate select the column by index.

Displaying Tables with Spring Shell

The output with the string is simple and can be enhanced. The *Spring Shell* project provides a way to draw tables. The method to transform a `Profile` object into a string with a table representation looks like this:

```
private String formatProfileAsTable( Profile p ) {
  TableModel tableModel = new TableModelBuilder<String>()
   .addRow().addValue( "ID" ).addValue( "" + p.getId() )
   .addRow().addValue( "Mane length" ).addValue( "" + p.getManelength() )
   .addRow().addValue( "Nickname" ).addValue( p.getNickname() )
   .addRow().addValue( "Birthdate" ).addValue( p.getBirthdate().toString() )
   .build();
  Table table =
   new TableBuilder( tableModel )
     .addFullBorder( BorderStyle.fancy_light ).build();
  return table.render( 100 );
}
```

Spring Shell type `TableModelBuilder`[10] allows the construction of `TableModel` objects, which then become a `Table`. This can be rendered into a string in the next step.

In `TableModelBuilder`, method `addRow()` introduces a new row, and new columns are added by `addValue(…)`. Our code creates four columns. Finally, the `build()` method returns a `TableModel` object, which is passed to `TableBuilder`. A border style is set, and `build()` returns the `Table` object, which `renders(100)` to the width of 100 characters in a `String`.

`TableBuilder` can do a little more. For example, headers can be added, and different border styles can be used. Method `formatProfileAsTable(…)` can be used with `map(…)`:

```
@ShellMethod( "Display profile" )
public String findprofile( long id ) {
  return Optional.ofNullable( em.find( Profile.class, id ) )
                 .map( this::formatProfileAsTable )
                 .orElse( "Unknown profile for ID " + id );
}
```

6.5.3 find(…) and getReference(…)

Besides the `find(…)` method, there is a method with identical parameter types that sound similar in name—`getReference()`:

```
<T> T getReference(Class<T> entityClass, Object primaryKey)
```

---

[10] *https://docs.spring.io/spring-shell/docs/current/api/org/springframework/shell/table/TableModelBuilder.html*

The key difference between `getReference(…)` and `find(…)` is that the former doesn't return a fully populated object. Instead, it returns a proxy, without triggering an SQL query by the Jakarta Persistence provider. The proxy employs lazy loading and waits until the first access to fill the entity bean.

The container remains unfilled until the proxy is accessed, and the Jakarta Persistence provider intercepts the method calls in the process. Thus, directly querying the instance variables doesn't trigger loading. The following code demonstrates this behavior:

```
Profile fillmoreFat = em.getReference( Profile.class, 1L );
System.out.println( fillmoreFat.getClass() );
System.out.println( fillmoreFat.getNickname() );
System.out.println( fillmoreFat.getLastseen() );
```

The `Class` object indicates the proxy type, and `getNickname()` triggers loading.

The `getReference(…)` method has two peculiarities. You could also say it has two weaknesses:

- If the entity doesn't exist, an `EntityNotFoundException` follows only later on the first access.
- The proxy is just an empty container at the beginning. There is no loading process until the proxy is accessed. Because the `EntityManager` can't know when code accesses the proxy, the database connection could have ended long ago. The result with Hibernate is an `org.hibernate.LazyInitializationException`.

We've seen two ways to load the entity beans, and both require the key. There are no more direct loading methods with the `EntityManager`—not even to get all entity beans, for example.

6.5.4 Query Options with the EntityManager

Queries via the `EntityManager` can be made in several ways:

- **`find(…)/getReference(…)`**
  The methods always return only one entity based on the primary key. There are no methods like `findAll()`.
- **JPQL**
  This query language can be thought of as OO SQL.
- **Native SQL**
  JPQL is at the level of perhaps SQL 92 and is therefore more than 30 years old. Databases have evolved massively, and therefore it's necessary to use Native SQL in some places to take full advantage of the databases' capabilities.
- **Criteria API**
  This introduces typed queries, and larger queries can be dynamically composed of small building blocks.

To submit JPQL or SQL, the `EntityManager` declares the following methods:

- `createQuery(…)` for JPQL queries
- `createNativeQuery(…)` for Native SQL queries

Two separate methods are needed because if `EntityManager` only gets a string, it can't read how to process the queries.

The queries can have placeholders. These are called *named queries*. These placeholders are written with a colon, just like the `NamedParameterJdbcTemplate`, so that they are easy to read.

6.6 Jakarta Persistence Query Language (JPQL)

As discussed earlier in this chapter, JPQL is modeled after SQL and includes familiar keywords. It provides support for the following:

- Query entities and their persistent states (`SELECT`)
- Updates (`UPDATE`)
- Delete instructions (`DELETE`)

In general, `UPDATE` and `DELETE` statements are used for bulk operations rather than updating or deleting individual entities.

JPQL doesn't include any DDL statements such as `CREATE TABLE`, `CREATE INDEX`, `DROP TABLE`, or `ALTER TABLE` that can be used to modify the structure and schema of the database.

Because a database can't execute JPQL queries directly, the Jakarta Persistence provider translates them into Native SQL. In the ideal scenario, the Jakarta Persistence provider intelligently selects the optimal SQL variant for each relational database management system (RDBMS).

6.6.1 JPQL Example with SELECT and FROM

We can examine a straightforward JPQL query that fetches all entities, which can be used to create a customized `findAll()` method:

```
SELECT p
FROM Profile AS p
```

The query starts with keyword `SELECT` if we want to read data. Next, after `FROM`, is the set of entities, the *domain*. In this case, the query retrieves results from the domain of profiles. An *identification variable* is defined after `AS`, which will prove useful later when applying constraints via the `WHERE` clause. For instance, the identification variable can be used to select profiles with a particular property.

The statement can be abbreviated because `AS` can be omitted:

```
SELECT p
FROM Profile p
```

Note

Hibernate defines queries with the *Hibernate Query Language* (*HQL*),[11] which is a precursor to JPQL. This allows the query to be further abbreviated so that it's simply called `from Profile`. However, the syntax is proprietary and shouldn't be used.

JPQL versus SQL

The primary contrast between JPQL and SQL is that SQL operates at the table level, dealing with relations, columns, and rows. In contrast, JPQL operates at the object model level, focusing on entities or objects and their persistent attributes. Consequently, JPQL results in not only a collection of columns but also objects or persistent attributes (see Table 6.2).

Query	Result List Types
`SELECT p FROM Profile p`	`com.tutego.date4u.core.profile.Profile`
`SELECT p FROM Photo p`	`com.….Photo`

Table 6.2 Typed Queries and Their Results

Thus, `SELECT p FROM Profile p` returns a collection of `Profile` objects, and `SELECT p FROM Photo p` returns `Photo` objects, but not simply column values.

Uppercase/Lowercase

For better distinction, JPQL keywords are capitalized in this book. However, the case of the keywords isn't relevant. Therefore,

```
select p from Profile p
```

gets the same result as

```
SELECT p FRoM Profile p
```

(The authors of Hibernate think lowercase identifiers are nifty.)

While the keywords can be case-sensitive, the same isn't true for the names of the listed entities and also for the identification variables and the persistent attributes. JPQL is a typed language, and the case of the entity names and persistent attributes is important (see Table 6.3).

---

[11] *https://docs.jboss.org/hibernate/orm/current//userguide/html_single/Hibernate_User_Guide.html#query-language*

JPQL	Mistake, Because . . .
`SELECT p FROM ProFile p`	It's called `Profile`, not `ProFile`.
`SELECT p FROM Profile P`	It's called `p`, not `P`.

Table 6.3 Errors in JPQL Queries

After `FROM`, there is no table name, but the name of the entity. When refactoring the entity name, we have to be careful that we also change the JPQL strings.

6.6.2 Build and Submit JPQL Queries with createQuery(…)

The entity bean is written, and we know simple JPQL queries, so now we can issue a query. This works in two steps: In the first step, we use the method `createQuery(…)`to build a query. The object construction doesn't initiate a database query. The data is retrieved in the second step with query methods such as `getResultList()`.

Here's an example where all profiles are to be determined and output:

```
TypedQuery<Profile> query =
    em.createQuery( "SELECT p FROM Profile p", Profile.class );
List<Profile> profiles = query.getResultList();
for ( Profile p : profiles )
  System.out.println( p );
```

With `createQuery(…)`, first the JPQL string is passed, and then a type token that identifies our entity bean. The result is an object of type `TypeQuery`,[12] which is generically typed with our `Profile`.

A call to the `getResultList()` method on the query loads the profiles we can look at.

The `createQuery(…)` method is overloaded with two variants:

- `TypedQuery<T> createQuery(String qlString, Class<T> resultClass)`
- `Query createQuery(String qlString)`

Interface `TypedQuery` is a subtype of `Query` (see Figure 6.5). `Query` lacks the generic type information, so `getResultList()` is also only able to return a `List<Object>`.

Tip: IntelliJ

IntelliJ IDEA Ultimate Edition detects errors in the JPQL strings and also offers keyboard completion. If the project is connected to a database, IntelliJ can also execute JPQL queries from the Java code.

---

[12] *https://jakarta.ee/specifications/persistence/3.1/apidocs/jakarta.persistence/jakarta/persistence/typedquery*

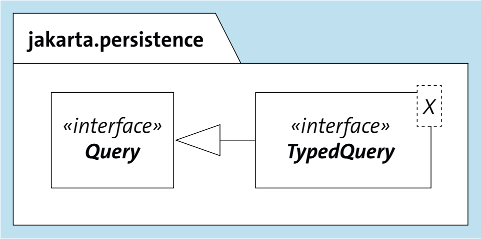


Figure 6.5 "TypedQuery": A Generically Declared "Query"

Restrict a Query

Because the result set can become large, it can be constrained using two methods:

- **`[Typed]Query setFirstResult(int startPosition)`**
  From which position should the results start? In other words, how many results at the beginning should be skipped?
- **`[Typed]Query setMaxResults(int maxResult)`**
  What is the maximum number of elements the collection should contain (after `setFirstResult(…)`)?

Because the methods return a `[Typed]Query`—depending on whether it was started with `createQuery(String)` or `createQuery(String, Class<T>)`—the calls can be cascaded well, like this:

```
profiles = query.setFirstResult( 1 )
                .setMaxResults( 2 )
                .getResultList();
```

The persistence provider converts JPQL to SQL. Usually, `setFirstResult(…)` is realized by `OFFSET`, and `setMaxResults(…)` by `LIMIT` or, alternatively, `FETCH`. For H2, the SQL looks like this:

```
select
    p1_0.id,
    …
    p1_0.nickname
from
    profile p1_0
offset ? rows fetch first ? rows only
```

This is useful for pagination, and we'll return to it at various points. I call this approach *OFFSET-LIMIT pagination* in the following. It's potentially slow, but easy to implement.

If you're only interested in the first element, the `getSingleResult()` method is handy. However, there must also be at least one element, or there will be a `NoResultException`.[13]

---

[13] This was discussed intensively: *https://github.com/eclipse-ee4j/jpa-api/issues/298*.

Task

We're looking for a new shell method in `EntityManagerCommands` that allows us to browse the list of all profiles page by page. The shell command should be applied like this:

```
page <page>
```

Here, `<page>` stands symbolically for the page index. The index starts at 0, and each page has 10 elements.

Proposed solution:

```
private final static int PAGE_SIZE = 10;
@ShellMethod( "Display profiles for a given page" )
public List<Profile> page( int page ) {
  return em.createQuery( "SELECT p FROM Profile p", Profile.class )
           .setFirstResult( page * PAGE_SIZE )
           .setMaxResults( PAGE_SIZE )
           .getResultList();
}
```

Stream Return

Besides the possibility to get the data directly as a `List`, `getResultStream()` is an alternative that returns a `Stream`. The result is a typed stream in the case of a `TypeQuery`. Here's an example:

```
TypedQuery<Profile> query = em.createQuery( "SELECT p FROM Profile p",
                                            Profile.class );
try ( Stream<Profile> stream = query.getResultStream() ) {
  stream.forEach( profile -> log.info( "{}", profile ) );
}
```

This stream can be run and consumed as known.

Note

The results can be filtered, sorted, or transformed with ease using stream operations. Nevertheless, it's important to recognize that heavy-duty operations shouldn't be performed by the Java side, as the database is better suited for such tasks.

6.6.3 Conditions in WHERE

The use of the `FROM` keyword is mandatory, and `SELECT` is employed when selecting entity beans. The straightforward syntax retrieves all entities from a specific domain. However, when dealing with vast datasets, it's uncommon to query all entity beans.

Typically, some constraints are required, and a `WHERE` clause can limit the result set. This `WHERE` clause can also be used for `UPDATE` and `DELETE` operations.

WHERE Clause and Identification Variable

Because criteria of the entity beans are usually used for restriction in a `WHERE` clause, the identification variable becomes important:

```
SELECT p FROM Profile p
```

The identification variable is `p`, and it helps us to use the `WHERE` clause to filter out exactly those entities that satisfy a certain condition.

Here's an example in which we're looking for all profiles that have a mane length greater than 10:

```
var jpql = "SELECT p FROM Profile p WHERE p.manelength > 10";
em.createQuery( jpql, Profile.class )
  .getResultList().forEach( profile -> log.info( "{}", profile ) );
```

The `WHERE` clause specifies the condition that every entity must meet.

6.6.4 Parameterize JPQL Calls

In query `SELECT p FROM Profile p WHERE p.manelength > 10`, the literal `10` is hard-coded in the string, but often the values come from user input. We've already discussed with SQL that concatenation with plus isn't good. Fortunately, there are fewer problems with a JPQL injection because JPQL can't contain DDL statements such as `DROP TABLE`. Still, it's not wise to concatenate JPQL strings because that floods the internal cache that manages JPQL strings.

The following is a *bad* example in which all profiles that match a pattern `name` are requested:

```
var name = …;
var jpql = "SELECT p FROM Profile p "
         + "WHERE  p.nickname LIKE '" + name + "'";
```

In JPQL, there is the same problem as in SQL—that the JPQL string is invalid at the end if the `name` contains a single quote character, for example. It's much better to use placeholders that will be filled later. The Jakarta Persistence API offers two possibilities for this:

- **Option A: Named parameter**

  ```
  SELECT p FROM Profile p WHERE p.nickname LIKE :name
  ```

- **Option B: Position parameters** (also called *index parameters*)

  ```
  SELECT p FROM Profile p WHERE p.nickname LIKE ?1
  ```

While named parameters are straightforward to read, position parameters are more suitable for Native SQL queries. A question mark followed by a number starting at 1 is used for position parameters, which is a convention commonly followed in SQL, where the indexing typically starts at 1 rather than 0.

Note

Named parameters and position parameters shouldn't be mixed.

Fill Parameters

For setting the parameters, [`Typed`]`Query` provides the overloaded methods `setParameter(…)`:

- `setParameter(String name, * value)`
- `setParameter(int position, * value)`

The methods assign a concrete value to a placeholder. The first parameter is either the name of the placeholder for named parameters or a position (index). The asterisk symbolizes different data types.

`TypedQuery`[14] declares the following `setParameter(…)` methods:

- `TypedQuery<X> setParameter(int position, Object value)`
- `TypedQuery<X> setParameter(int position, Calendar value, TemporalType temporalType)`
- `TypedQuery<X> setParameter(int position, Date value, TemporalType temporalType)`
- `TypedQuery<X> setParameter(Parameter<Calendar> param, Calendar value, TemporalType temporalType)`
- `TypedQuery<X> setParameter(Parameter<Date> param, Date value, TemporalType temporalType)`
- `<T> TypedQuery<X> setParameter(Parameter<T> param, T value)`
- `TypedQuery<X> setParameter(String name, Object value)`
- `TypedQuery<X> setParameter(String name, Calendar value, TemporalType temporalType)`
- `TypedQuery<X> setParameter(String name, Date value, TemporalType temporalType)`

Because the methods provide a `TypedQuery`, the expressions can be written nicely in a cascaded fashion.

---

[14] *https://jakarta.ee/specifications/persistence/3.1/apidocs/jakarta.persistence/jakarta/persistence/typedquery*

Let's rewrite the previous example with the "wrong" concatenation, and now use named parameters and `setParameter(…)`:

```
var name = "Fat";
var jpql = "SELECT p FROM Profile p WHERE p.nickname LIKE :name";
em.createQuery( jpql, Profile.class )
  .setParameter( "name", "%" + name + "%" )
  .getResultList()
  .forEach( profile -> log.info( "{}", profile ) );
```

The query returns all profiles where the nickname contains the substring `name`. When querying substrings, the substring is framed with `%` characters as in SQL.

Note

A common mistake is to set the percent signs in the JPQL string. But it's `LIKE :name` and not `LIKE %:name%`, and it's `"%" + name + "%"` and not `name` when setting the parameter. Intellij Ultimate Edition shows the error.

Let's look at a second scenario:

```
var jpql = "SELECT p FROM Profile p WHERE p.id = :id";
var id = 1;
log.info( "{}",
          em.createQuery( jpql, Profile.class )
            .setParameter( "id", id )
            .getSingleResult() );
```

The ID is used to select a profile. Even though this query seems nonsensical in its structure, as the `find(…)` method in the `EntityManager` serves that purpose, it effectively demonstrates an issue. As there will only be one element, `getSingleResult()` is suitable, including the exception if there is no row for the ID. At first glance, the code appears to be correct, but is it truly so? No. There will be an error, and it will be an exception with a message:

```
Parameter value [1] did not match expected type [java.lang.Long (n/a)]
```

The problem is that autoboxing makes call `setParameter(Integer.valueOf(1))`, but the ID column is of type `Long`. This is tricky, and therefore caution is required to use the appropriate types correctly. The correct code is

```
var id = 1L;
…
query.setParameter( "id", id );
```

or, more explicitly,

```
query.setParameter( "id", Long.valueOf( id ) );
```

Task

A new shell method is to be written in the `EntityManagerCommands` class that returns a list of profiles seen in the past six months. The entity class `Profile` has a persistent attribute `lastseen`.

Proposed solution:

```
@ShellMethod( "Display latest seen profiles" )
public List<Profile> lastseen() {
  return
    em.createQuery( "SELECT p FROM Profile p WHERE p.lastseen > :lastseen",
                    Profile.class )
      .setParameter( "lastseen", LocalDateTime.now().minusMonths( 6 ) )
      .getResultList();
}
```

Listing 6.7 EntityManagerCommands.java Extension

Because we want to search for a timestamp, `setParameter(…)` is possible in principle with a `Date` or `Calendar` object. But because `LocalDate` is also possible, we don't have to work with the "old" data types.

6.6.5 JPQL Operators and Functions

The provided examples have demonstrated the effectiveness of prepared statements for secure and optimized database queries. However, the covered operators have been limited, with only the greater than operator and `LIKE` having been explored so far. To broaden our understanding, let's explore some additional operators that can be used in SQL queries.

Arithmetic and Logical Operators

In the `WHERE` clause, we've worked with the comparison so far, but the JPQL also provides the following:

- Arithmetic operators: +, - (unary operators), *, /, +, -
- Logical operators: `NOT`, `AND`, `OR`

Logical operators link truth values. For the remainder, as we'll see later, there is a function.

Comparison Operators

The comparison operators `=`, `<>` (unequal), `>`, `>=`, `<`, and `<=` are available. The character for unequal differs from Java; it's written with a less-than sign.

Here are some examples to consider. Suppose we want to find all profiles that have a nonzero mane length:

```
SELECT p FROM Profile p WHERE manelength <> 0
```

The query can be rewritten. It's conceivable to check for mane length equal to 0 and negate the statement with `NOT`:

```
SELECT p FROM Profile p WHERE NOT (manelength = 0)
```

The logical operator `AND` can be used to connect truth values. For instance, assuming that we want to retrieve all `Profile`s with a mane length greater than 10 and less than 20 (inclusive), we can use the `BETWEEN` operator:

```
SELECT p FROM Profile p WHERE manelength > 10 AND manelength < 20
```

Here's an example with `OR`: we're looking for profiles that have the nickname "FillmoreFat" or "DanSing":

```
SELECT p FROM Profile p WHERE nickname = 'FillmoreFat' OR nickname='DanSing'
```

[NOT] BETWEEN Expressions

For range queries, there is `BETWEEN` and the negation `NOT BETWEEN`. `x BETWEEN y AND z` is semantically equivalent to `y <= x AND x <= z`. The operator `BETWEEN` compares not only numbers but also dates and strings.

Here are some examples:

```
SELECT p FROM Profile p WHERE birthdate BETWEEN '1980-01-01' AND '2019-01-01'
SELECT p FROM Profile p WHERE manelength BETWEEN 10 AND 20
SELECT p FROM Profile p WHERE nickname NOT BETWEEN 'A' AND 'H'
```

When specifying date-time values, care must be taken to select the notation of the JDBC driver. The JPQL recommends notations with curly brackets, for example, `{TS}` for a timestamp. However, this may not be processed by the JDBC driver.

[NOT] IN Expression

Currently, when column contents are to be compared with certain values, `IN` can be used. Here's an example: the profiles with the mane lengths 1, 2, or 10 are searched for:

```
SELECT p FROM Profile p WHERE p.manelength IN (1, 2, 10)
```

Here's a second example: we're looking for exactly those profiles where `attractedToGender` is set to 1 or 2:

```
SELECT p FROM Profile p WHERE p.attractedToGender IN (1, 2)
```

In the end, the `IN` is nothing more than a notation with `OR`, only shorter.

Just as in SQL, subqueries can be formulated in JPQL, which then restrict the set.

IS [NOT] NULL

In SQL, `NULL` holds a special status. JPQL allows us to check whether specific columns and persistent attributes are `NULL` or not. In the `attractedToGender` attribute, we set the value to `NULL` to indicate that a profile is interested in all genders. Therefore, we can use `IS NULL` to determine which profiles have an interest in all genders or the opposite, by using `IS NOT NULL`:

```
SELECT p FROM Profile p WHERE p.attractedToGender IS NULL
SELECT p FROM Profile p WHERE p.attractedToGender IS NOT NULL
```

[NOT] LIKE with Pattern Matching

The `LIKE` operator enables pattern queries on strings, while `NOT LIKE` signifies negation. Two escape characters are used:

- `_` matches exactly one arbitrary character.
- `%` represents a string, including an empty one.

For instance, the following queries illustrate the usage of `LIKE`.

To retrieve all profiles whose nicknames start with “A”:

```
SELECT p FROM Profile p WHERE nickname LIKE 'A%'
```

To fetch all profiles with a nickname length of seven characters:

```
SELECT p FROM Profile p WHERE nickname LIKE '_______'
```

The SQL standard doesn’t offer many string comparison options. The capability to handle regular expressions is specific to the RDBMS in use. Some examples include MySQL, MS SQL, Oracle, and PostgreSQL.

Functions for Strings

JPQL isn’t limited to the use of built-in operators only; it also provides built-in functions. These functions are listed in the specification, and their syntax is used to invoke them.

The following string functions return strings:

- `CONCAT(string_expression, string_expression {, string_expression}*)`
- `SUBSTRING(string_expression, arithmetic_expression [, arithmetic_expression])`
- `LOWER(string_expression)`, `UPPER(string_expression)`
- `TRIM([[trim_specification] [trim_character] FROM] string_expression)`, where `trim_specification ::= LEADING | TRAILING | BOTH`

String functions that return length and location are as follows:

- `LENGTH(string_expression)`
- `LOCATE(string_expression, string_expression[, arithmetic_expression])` (index starts at 1, and 0 means not found)

`CONCAT` concatenates strings. This also allows persistent attributes to be queried and concatenated. A nice example is concatenating first names with a last name separated by a space. The other functions are self-explanatory.

Here are three examples:

- We're looking for profiles whose nicknames begin with "A":

  ```
  SELECT p FROM Profile p WHERE SUBSTRING(p.nickname, 1, 1) = 'A'
  ```

  The expression extracts the character at location 1 and compares it to A.
- In which profile is there text with more than 40 characters?

  ```
  SELECT p FROM Profile p WHERE LENGTH(p.description) > 40
  ```

- What profiles have nicknames that start with "Fillmore" in the first place?

  ```
  SELECT p FROM Profile p WHERE LOCATE('Fillmore', p.nickname) = 1
  ```

Arithmetic Functions

Although JPQL includes operator characters such as +, -, *, and /, the percent sign (%) isn't a valid operator in JPQL. Instead, JPQL employs the built-in `MOD` function for this purpose. JPQL offers a limited number of arithmetic functions overall:

- **`ABS(arithmetic_expression)`**: Absolute value
- **`SQRT(arithmetic_expression)`**: Root to base 2
- **`MOD(arithmetic_expression, arithmetic_expression)`**: Remainder

Here's an example: we're looking for the profiles that have a mane length defined that is divisible by 10, such as 0, 10, 20:

```
SELECT p FROM Profile p WHERE MOD(p.manelength, 10) = 0
```

New Numeric Functions in Jakarta Persistence 3.1

Spring Boot 3.0 includes a Hibernate ORM version 6.1, so features from Jakarta Persistence 3.1 are possible. New functions have been added, the names of which are self-explanatory:

- `CEILING(arithmetic_expression)`
- `EXP(arithmetic_expression)`
- `FLOOR(arithmetic_expression)`
- `LN(arithmetic_expression)`
- `POWER(arithmetic_expression, arithmetic_expression)`
- `ROUND(arithmetic_expression, arithmetic_expression)`
- `SIGN(arithmetic_expression)`

Functions for Time and Date

There are functions that provide the server time: `CURRENT_DATE`, `CURRENT_TIME`, and `CURRENT_TIMESTAMP`. The returns are `java.sql.Date`, `java.sql.Time`, and `java.sql.Timestamp`. In Jakarta Persistence 3.1, the following functions are new: `LOCAL DATE`, `LOCAL DATETIME`, and `LOCAL TIME`. The returns are the modern data types `java.time.LocalDate`, `java.time.LocalDateTime`, and `java.time.LocalTime`. They are useful for queries to determine whether a date or time is in the past or future.

It's important to note that the date and time are coming from the database server, not from the Jakarta Persistence provider (i.e., the client side).

Another new function in Jakarta Persistence 3.1 is `EXTRACT`. This function can extract segments of a date and/or time value, which can look like this:

```
EXTRACT(YEAR FROM p.birthdate)
```

The segment to be extracted is listed first. Allowed segments are `YEAR`, `QUARTER`, `MONTH`, `WEEK`, `DAY`, `HOUR`, `MINUTE`, and `SECOND`.

6.6.6 Order Returns with ORDER BY

Thus far, we've explored various queries and used two essential JPQL keywords: `FROM` specifies the domain, while `SELECT` selects the entities. In principle, the order in which entities are returned is unspecified.

JPQL provides the `ORDER BY` keyword to specify an ordering criterion, as in SQL. For example, if we want to sort all profiles by mane length, we can write it like this:

```
SELECT   p
FROM     Profile p
ORDER BY p.manelength
```

By default, sorting in JPQL is in ascending order, meaning small values appear first, and large values appear last. Explicitly stating `ASC` after the name of the persistent attribute specifies *ascending* order. To sort in *descending* order, `DESC` is used instead.

Multiple sorting criteria are separated by commas. For example, if two profiles have the same mane length, we can use another sorting criterion, such as the nickname, like this:

```
SELECT   p
FROM     Profile p
ORDER BY p.manelength DESC, p.nickname
```

In this example, the profiles are sorted in descending order by mane length and in ascending order by nickname if the mane lengths are equal.

6.6.7 Projection on Scalar Values

For queries of the type `SELECT e FROM Entity e`, the result is a collection of entity objects. This is also the default for the Jakarta Persistence API, but other return types are possible. There are `SELECT` queries that don't return entities:

- With `SELECT`, we select the objects or persistent attributes that the result list should contain. This is a projection on a part of the entity states.
- Aggregate functions, such as `COUNT(…)`, return only one numeric value.

With `SELECT` statements, specific persistent attributes can be selected, such as strings, wrapper objects, temporal types, and so on, as well as `COUNT` with aggregate functions, for example. The syntax for `SELECT-FROM` remains essentially the same.

In the next example, only the nickname of all profiles is requested as a result, sorted by `lastseen`, and no complete `Profile` objects:

```
var jpql = "SELECT p.nickname FROM Profile p ORDER BY p.lastseen";
em.createQuery( jpql, String.class )
  .getResultList()
  .forEach( log::info );
```

This is a typical example of projecting a value. `SELECT p.nickname` specifies that we want to select the `nickname` attribute of all `Profile` entities. `p` is still the alias for the `Profile` entity, and dot notation is used to access the `nickname` attribute. The sorting remains the same, and there are no changes in the rest of the query.

The result doesn't contain `Profile` objects anymore. Because `nickname` is a `String`, we pass type token `String.class` instead of `Profile.class` in the `createQuery(…)` method. The result of the query is a list of `Strings`.

Note

It's crucial to distinguish between projections and entity returns. When the result is an entity bean, the `EntityManager` takes care of managing the object as a *managed entity bean*. However, this isn't the case with projections. In Section 6.9, we'll delve into this topic further.

DISTINCT

When using projections in JPQL, it's possible to retrieve multiple instances of the same value, such as when selecting the mane length attribute with the query `SELECT p.manelength FROM Profile p`. To eliminate duplicate values from the result set, the `DISTINCT` keyword can be used in JPQL. The following JPQL expression removes all duplicate mane lengths so that each mane length occurs only once:

```
SELECT DISTINCT p.manelength FROM Profile p
```

6.6.8 Aggregate Functions

We've already talked about aggregate functions such as `COUNT(*)`. JPQL, like SQL, supports a number of aggregate functions:

- **`MIN(…), MAX(…)`**
  Returns the smallest/largest value of a collection of numeric values, strings (lexicographic comparison), or date values. The return corresponds to the type of the persistent property.
- **`SUM(…)`**
  Returns the sum of numeric values depending on the type as `Double`, `BigInteger`, or `BigDecimal`.
- **`AVG(…)`**
  Returns the arithmetic mean of numeric values as a `Double`.
- **`COUNT(…)`**
  Returns as `Long` the number of all elements in the query.

The responses are always in the form of objects, including wrapper objects. Any `NULL` values are automatically excluded before calling the relevant aggregate functions. For instance, if a column value is `NULL` during a summation, it will be disregarded.

Let's look at four examples:

- The average mane length is required:

  ```
  SELECT AVG(p.manelength) FROM Profile p
  ```

Calling the aggregate function `AVG(p.manelength)` returns the arithmetic mean as a floating-point number.[15]

- What is the date of the last viewed profile?

  ```
  SELECT MAX(p.lastseen) FROM Profile p
  ```

  The aggregate function `MAX(p.lastseen)` returns the largest date. It's important to remember that it's not the profile that is returned, but a timestamp.

- How many profiles have `fat` in the name?

  ```
  SELECT COUNT(p) FROM Profile p WHERE LOWER(p.nickname) LIKE '%fat%'
  ```

  The fragment `LIKE '%fat%'` tests whether the persistent attribute contains substring `fat`. Before that, the column contents are converted to lowercase with the `LOWER` function.

- To find out how many different mane lengths there are in the `Profile` entity, we can use the following JPQL query:

  ```
  SELECT COUNT(DISTINCT p.manelength) FROM Profile p
  ```

6.6.9 Projection on Multiple Values

We've learned that in JPQL, it's possible to project on persistent attributes instead of selecting the complete entity. This can be achieved by specifying the names of the persistent attributes after the `SELECT` keyword. For example, if we want to project only the `nickname` attribute of the `Profile` entity, we can write `SELECT p.nickname FROM Profile p`. The result of this query will be a list of strings containing the nicknames.

It's also possible to project multiple persistent attributes at once; for example, if we want to retrieve both the nickname and the mane length of each profile, we can write the following:

```
SELECT p.nickname, p.manelength FROM Profile p
```

While in the first case with `p.nickname`, the result of `getResultList()` is a list of strings, two and more values return small `Object` arrays. This also means that the type information is lost at translation time. Let's look at an example:

```
var jpql = "SELECT p.nickname, p.manelength FROM Profile p";

TypedQuery<Object[]> query = em.createQuery( jpql, Object[].class );
List<Object[]> nicknamesAndManelength = query.getResultList();

for ( Object[] columns : nicknamesAndManelength )
  System.out.println( columns[ 0 ] + " " + columns[ 1 ] );
```

---

[15] On the SQL side, `AVG` returns the result of the same type as the column defined, for example, `Integer` for an integer column. Therefore, it may be surprising if the floating-point number is missing the decimal values. The behavior may differ depending on the persistence provider.

In `createQuery(…)`, the type token `Object[]` is passed, and there is a `List<Object[]>`at the end. The small object arrays carry the information on the two persistent attributes for the concrete nickname and the mane length.

A disadvantage with this approach is that the type information is completely lost. We'll discuss a solution shortly, where instead of `Object[]`, there is a separate typed container.

Type Tuple

Jakarta Persistence offers an alternative to the `Object` array via the `Tuple`[16] data type (see Figure 6.6).

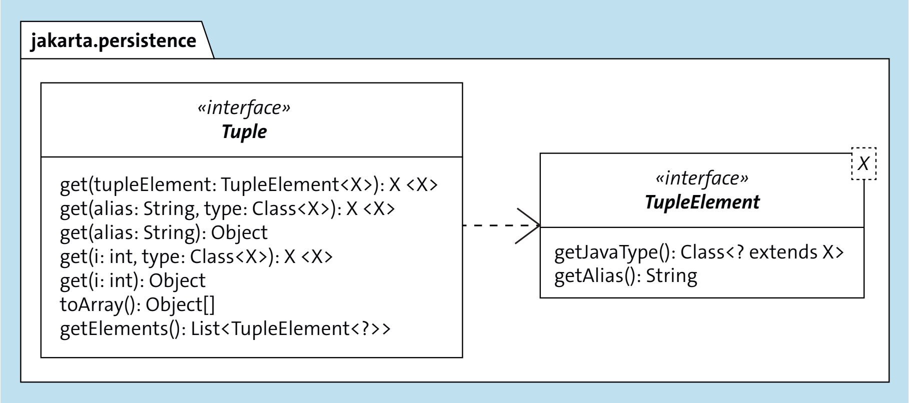

jakarta.persistence
«interface»
Tuple
get(tupleElement: TupleElement<X>): X <X>
get(alias: String, type: Class<X>): X <X>
get(alias: String): Object
get(i: int, type: Class<X>): X <X>
get(i: int): Object
toArray(): Object[]
getElements(): List<TupleElement<?>>
«interface»
TupleElement
X
getJavaType(): Class<? extends X>
getAlias(): String

Figure 6.6 "Tuple" Data Type

A `Tuple` is an association between the persistent attribute and the respective associated value. Let's look at an example. Again, `nickname` and `manelength` are to be requested from all profiles:

```
var jpql = "SELECT p.nickname, p.manelength FROM Profile p";

TypedQuery<Tuple> query = em.createQuery( jpql, Tuple.class );
List<Tuple> nicknames = query.getResultList();

for ( Tuple tuple : nicknames )
  log.info( "Nickname={}, mane length={}",
            tuple.get( 0 ), tuple.get( 1, Short.class ) );
```

In `createQuery(…)`, the type token `Tuple.class` appears and no longer `Object[].class`. Instead of an array with two elements, we get back a `Tuple` object, which is a small associative data structure that stores both persistent attributes for one row. The UML diagram shows the methods, and we can access the first element (`nickname`) with `tuple.get(0)` and `manelength` with `tuple.get(1)`. With the generic `get*(…)` methods, a type token can be passed, so that the return type is no longer `java.lang.Object`, but self-determined. Of course, these types must also match.

---

[16] *https://jakarta.ee/specifications/persistence/3.1/apidocs/jakarta.persistence/jakarta/persistence/tuple*

Grouping and Filtering with GROUP BY and HAVING

Like SQL, the JPQL supports grouping with `GROUP BY` and the filtering of groups with `HAVING`. For example, `GROUP BY` can be used to answer the question of how often which mane length occurs:

```
var jpql = """
    SELECT p.manelength, COUNT(p)
    FROM Profile p
    GROUP BY p.manelength
    ORDER BY p.manelength DESC""";

TypedQuery<Tuple> query = em.createQuery( jpql, Tuple.class );
List<Tuple> tuples = query.getResultList();

for ( Tuple tuple : tuples )
  log.info( "mane length={}, count={}", tuple.get( 0 ), tuple.get( 1 ) );
```

The attachment `GROUP BY p.manelength` will form groups of profiles with the same mane length, and from the many small groups, the aggregate function `COUNT` will provide the number at the end. `SELECT p.manelength, COUNT(p)` is a projection, and the results can be collected as `Object[]` or `Tuple`. The entire list should be sorted by the database, which is taken care of by `ORDER BY p.manelength DESC`. The descending sorting would put the largest mane length at the front and the smallest at the back.

`HAVING` removes entire groups from the results. This is particularly interesting for aggregate functions such as `AVG`, which determine average values for a group. With `HAVING`, you can then remove the entire group if the average values don't have the desired property.

Constructor Expressions in SELECT

We've seen so far that a query can produce different results:

- List of entities
- List of scalar values
- List of `Tuple` objects
- List of `Object` arrays

If the queries return entities or scalar values, they can be typed using `TypedQuery`. With projection, the results can be collected as `Tuples` or `Object` arrays. This isn't optimal because type information is missing. Later, type conversions have to be done explicitly in the code, and this in turn increases the risk of `ClassCastException`s.

The Jakarta Persistence offers a type-safe alternative with *constructor expressions*:

```
SELECT
 new com.tutego.boot.jpa.demo.ProfileValueObject(p.nickname, p.manelength)
FROM Profile p
```

Behind `new` is a *fully qualified class name*, and that leads to the call of the parameterized constructor. What was previously comma-separated directly after `SELECT` now forms the arguments for a constructor call.

`ProfileValueObject` is a container, and because Jakarta Persistence only calls the constructor but not the setters, a record is also possible:

```
public record ProfileValueObject(
  String name,
  short manelength
) {
  @Override public String toString() {
    return name + " has mane length " + manelength;
  }
}
```

It doesn't matter which objects the JPQL expression maps to; we can also take existing classes. The only important thing is that they have a parameterized constructor. In addition, `java.awt.Point` is conceivable if the selected values are numeric. The names of the parameter variables are unimportant.

If you build the query with `createQuery(…)`, `ProfileValueObject.class` is passed as the type token. A complete example looks like this:

```
var jpql = """
 SELECT
  new com.tutego.boot.jpa.demo.ProfileValueObject(p.nickname, p.manelength)
 FROM Profile p""";

em.createQuery( jpql, ProfileValueObject.class )
  .getResultList()
  .forEach( profile -> log.info( "{}", profile ) );
```

The result of this query is a list of `ProfileValueObject` objects.

Constructor expressions offer the benefit of being well-suited for projection when only specific aspects of an entity are pertinent, while also providing excellent type safety. Records, being immutable types, don't imply any modifications to the persistence context. They are typically not managed entity beans, which proves advantageous when constructing a large set of containers as it saves memory by freeing up the

`EntityManager` from having to manage the objects. Further details on this topic are covered in Section 6.9.

6.6.10 Named Declarative Queries (Named Queries)

When using the `EntityManager`, it's possible to send a JPQL query. However, it's not advisable to scatter JPQL strings throughout the source code because any changes made to the entity bean often require corresponding modifications to the JPQL. If JPQL is distributed across different parts of the program, locating and rectifying errors can be difficult and time-consuming. It's best to keep such internal details, such as table structure, hidden from clients. With Jakarta Persistence, JPQL queries can be associated with the entity bean itself and given a name for later reference. This functionality is called a *named query*.

Declare a Named Query with @NamedQuery

A named query is defined at the entity bean using `@NamedQuery`:[17]

```
@Entity
@NamedQuery( name  = "Profile.findAll",
             query = "SELECT p FROM Profile p" )
public class Profile { … }
```

The JPQL string is stored under a name so that the client can fetch the query later. The query is pooled with queries from other beans that are in the same session.

Tip

To avoid naming conflicts, a hierarchical naming is recommended; usual it's *entity name + "." + JPQL query*. This is also how our example does it. Whether the separator is a dot, a minus sign, or a double cross doesn't matter. For Spring Data JPA, there is a scenario where it must be a dot (for details, see Chapter 7, Section 7.6.8).

Extract a Query

The `EntityManager` provides method `createNamedQuery(…)` to retrieve a named query registered by `@NamedQuery`. It may look like this:

```
TypedQuery<Profile> query =
    em.createNamedQuery( "Profile.findAll", Profile.class );
query.getResultList()
     .forEach( profile -> log.info( "{}", profile ) );
```

---

[17] *https://jakarta.ee/specifications/persistence/3.1/apidocs/jakarta.persistence/jakarta/persistence/namedquery*

Instead of writing the query at this place, the client uses a logical name for the query. When using `createNamedQuery(…)`, the type token is still passed, and the return is a `TypedQuery`, which then proceeds as usual. There is also an untyped `Query` variant, `createNamedQuery(String name)`.

By using named queries, we've successfully hidden the JPQL string one level further. This means that it doesn't appear in the client code but is located centrally at the entity bean. If any changes are made to the entity bean, it's easy to check the named queries to see if any modifications to the JPQL strings are necessary.

Multiple @NamedQuery and @NamedQueries

It's not uncommon for more than one named query to be pinned to the entity bean. The `@NamedQuery` annotation is *repeatable*:

```
@NamedQuery( name = "…", query = "…" ),
@NamedQuery( name = "…", query = "…" )
```

In the earlier specifications, the annotation `@NamedQuery` wasn't yet repeatable, and therefore in slightly older programs, you can find a special container of type `@NamedQueries`:

```
@NamedQueries( {
  @NamedQuery( name = "…", query = "…" ),
  @NamedQuery( name = "…", query = "…" )
} )
```

It's no longer necessary.

Named Query with Parameter

A JPQL statement can contain placeholders for parameters—and this is also true for a named query. The notation is identical. Here's an example:

```
@NamedQuery(
  name  = "Profile.findByNickname",
  query = "SELECT p FROM Profile p WHERE p.nickname = :nickname"
)
```

This named query is called `Profile.findByNickname`. The JPQL selects all profiles but restricts them by nickname.

For an execution, `createNamedQuery(…)` will return a [Type]`Query` object, and `setParameter(…)` sets the named parameter, in our example with the nickname. `getResultList()` returns the result. (Because the nickname is `UNIQUE` in SQL, the result can only be none or one `Profile`. The function `getSingleResult()` can't be used because it doesn't return `null` if there are no results, but a `NoResultException`).

In code, it looks like this:

```
TypedQuery<Profile> query =
    em.createNamedQuery( "Profile.findByNickname", Profile.class );
query.setParameter( "nickname", "FillmoreFat" )
     .getResultList().forEach( profile -> log.info( "{}", profile ) );
```

Of course, without an assigned parameter value for `nickname`, there will be an error during execution.

6.7 Call Database Functions and Send Native SQL Queries

Although JPQL provides a useful abstraction over SQL queries, it has its limitations. For instance, it can't handle tasks that require linguistic similarities in the search for a name. For example, the last name "Ullenboom" may be spelled with one "l" or with "bohm" instead of "boom", which is a common occurrence for surnames such as Meyer, where the spelling is ambiguous.

One approach to solving this problem is to use the *Soundex code*, which assigns a code to a name based on its sound. For example, the Soundex code for "Ullenboom" is U451. In addition, "Uhlenboom", "Ullenbohm", and "Uhlenbohm" all have the same Soundex code. Most databases provide a `SOUNDEX` function for calculating the Soundex code, which can be used for comparison.

While JPQL doesn't define a `SOUNDEX` function, it's possible to call arbitrary database functions from JPQL, and even submit Native SQL queries when necessary. Jakarta Persistence provides two ways to send Native SQL queries:

- SQL queries can be passed to the `EntityManager` method `createNativeQuery(…)`.
- SQL can be attached to an entity bean with a logical name via the `@NamedNativeQuery` annotation, which is similar to `@NamedQuery`. This annotation is powerful and allows for the transfer of columns to objects.

Using these features, we can handle cases where the Jakarta Persistence provider might not generate the best SQL queries.

6.7.1 Call Database Functions: FUNCTION(…)

If JPQL doesn't provide a function, but the database does, you can call a predefined or user-defined database function with `FUNCTION(…)`. For our example with the `SOUNDEX` function, it can look like this:

```
SELECT p
FROM   Profile p
WHERE  FUNCTION('SOUNDEX', nickname) = FUNCTION('SOUNDEX', ?1)
```

This limits portability, but you don't have to leave JPQL to use database-specific functions.

6.7.2 Use createNativeQuery(...) via EntityManager

Among the reasons why JPQL can't be used is the lack of specific SQL capabilities. This doesn't have to be something as simple as a `SOUNDEX` function, but let's stick with the example and modify the name a bit:

```
var search = "filmorefart";
var sql = """
          SELECT *
          FROM Profile
          WHERE SOUNDEX(nickname) = LOWER(SOUNDEX(?)""";

@SuppressWarnings( "all" )
List<Profile> results = em.createNativeQuery( sql, Profile.class )
                          .setParameter( 1, search )
                          .getResultList();
results.forEach( profile -> log.info( "{}", profile.getNickname() ) );
```

The SQL query is to select all columns of the `Profile` table. By the `WHERE` clause, only the rows whose Soundex code from the nickname is equal to the Soundex code from the searched string come into the result set.[18]

For native methods, the `create`**`Native`**`Query(…)` method is used instead of `createQuery(…)`. Two different methods are necessary because the Jakarta Persistence provider can't read from the string whether it's a Native SQL or JPQL query. The parameterization of the methods is the same. It's a bit unfortunate that `createNativeQuery(…)` only returns the base type `Query`, no `TypedQuery`, which is why generic type information is missing in the following. From `Query`, the parameter is set; that is, the question mark is filled with the search term. While you can work with `:name` for named parameters, this isn't necessarily supported for native queries and depends on the Jakarta Persistence provider.

When parameterizing the query, `getResultList()` returns a raw type `List`. It's cheating if we make a `List<Profile>` out of the `List`, but this is possible—only the compiler must be quieted with `@SuppressWarnings( "all" )`.

The Soundex code is limited in its similarity search and also only considers English names. For Chinese, Japanese, or Korean, it would make no sense at all. Moreover, our solution is much too slow in practice; for performant searches, we would store the Soundex code in a separate column as well. But all this is beside the point for us because we learned two things: how to use native queries, and how to use the Soundex code.

Using the `createNativeQuery(…)` method has two disadvantages:

---

[18] Actually, the `SOUNDEX` comparison should be case-insensitive. But this is a bug that will be fixed in the next version.

- There are SQL queries in the source code. We had previously encountered named queries with JPQL. These would also be handy with Native SQL.
- Our SQL selects all columns, and “by chance” they match the persistent attributes of the entity bean. But what if the columns and the persistent attributes of an entity bean don’t match at all?

6.7.3 @NamedNativeQuery

When working with Native SQL queries, it’s better to use `@NamedNativeQuery` instead of `createNativeQuery(…)`. Similar to `@NamedQuery`, which is used for JPQL statements, `@NamedNativeQuery` associates a logical name with a SQL string within an entity bean. This approach ensures that the native queries are confined to the entity bean class, preventing the SQL strings from being scattered throughout the source code. If any modifications to the SQL are necessary, they can be made at a single point in the code.

`@NamedNativeQuery` also has the `name` and `query` attributes:

```
@NamedNativeQuery(
  name = "Profile.findFuzzyNickname",
  query = """
          SELECT * FROM profile
          WHERE SOUNDEX(nickname)=SOUNDEX(?)""",
  resultClass = Profile.class
)
```

The annotation attribute `name` determines the name of the query, the attribute `query` determines the SQL string. In addition, the specification of `resultClass` is necessary if the result is to be mapped to entity bean `Profile`. If no entity is to be created at the end, annotation attribute `resultClass` can be omitted.

Note

The Jakarta Persistence provider may need two SQL queries for the mapping, so the performance isn’t as optimal as expected.

@NamedNativeQuery: Call via createNamedQuery(…)

The query declared with `@NamedNativeQuery` can be executed using the `EntityManager` method `createNamedQuery(…)`. This may come as a surprise, as we’ve seen the method used earlier to submit named JPQL queries. However, the Jakarta Persistence provider recognizes from the name that it’s a named native query. Here’s an example of execution:

```
em.createNamedQuery( "Profile.findFuzzyNickname", Profile.class )
  .setParameter( 1, "filmorefaat" )
  .getResultList()
  .forEach( profile -> log.info( "{}", profile.getNickname() ) );
```

Setting the parameter, sending, and running are done as shown previously.

6.7.4 @NamedNativeQuery with resultSetMapping

Native queries offer the advantage of being able to execute arbitrary SQL queries without the need for the JDBC API. Furthermore, the Jakarta Persistence provider can map the results to any custom object, not necessarily an entity bean.

As described under Section 6.6.9, in the constructor expressions in `SELECT` projection with JPQL to the `ProfileValueObject` container, the data type was a record:

```
public record ProfileValueObject(
  String name,
  short manelength
) { }
```

We want to take up the example again.

For practice, we'll now program an example using Native SQL to retrieve `ProfileValueObjects` from the database. The query will be slightly different from usual, as we only want to retrieve profiles with a nickname that contains two vowels in a row, making it sound odd. To achieve this, we'll use the `REGEXP_LIKE` function, which is available in many database management systems and performs a regular expression (regex) match to check if a column value matches a regex. We can name this query `Profile.containsTwoVowelsNickname`:

```
@NamedNativeQuery(
    name = "Profile.containsTwoVowelsNickname",
    query = """
            SELECT nickname, manelength
            FROM profile
            WHERE REGEXP_LIKE(nickname, '[aeiou]{2}', 'i')""",
    resultSetMapping = "nicknameManelengthMappingToProfileValueObject" )
```

The SQL query demonstrated uses the regular expression `[aeiou]{2}` to match two vowels following each other and the flag `'i'` to ignore case. The resulting query selects the columns `nickname` and `manelength`, and the mapped result is the `ProfileValueObject`.

To map the `@NamedNativeQuery` to any object, a `resultSetMapping` is needed. A `ResultSet` is the JDBC type for the results, and the annotation attribute name tells what it's about: mapping the results to a target object. `resultSetMapping` names the mapper, which is declared elsewhere via a second annotation `@SqlResultSetMapping`.

@SqlResultSetMapping

`@SqlResultMapping` describes the mapping. The basic structure follows:

```
@NamedNativeQuery( … )
@SqlResultSetMapping(
    name = "nicknameManelengthMappingToProfileValueObject",
    …
)
@Entity
public class …
```

The `@SqlResultSetMapping` annotation is added to the entity bean class to map the results of the Native SQL query to `ProfileValueObject`. It may seem odd that the mapping isn't declared within `@NamedNativeQuery`, but this approach enables reusability. The same mapping can be used for different native queries, avoiding the need to repeat the mapping details for each `@NamedNativeQuery`. By giving the mapping a logical name, it can be referenced by different queries. The name is user-defined.

Attributes of @SqlResultSetMapping

We've seen the annotation attribute `name` of `@SqlResultSetMapping`; three annotation attributes are added:

```
@SqlResultSetMapping
@Repeatable(SqlResultSetMappings.class)
@Target({TYPE})
@Retention(RUNTIME)
public @interface SqlResultSetMapping {

    String name();

    EntityResult[] entities() default {};       ❶
    ConstructorResult[] classes() default {}; ❷
    ColumnResult[] columns() default {};        ❸

}
```

The three annotation attributes `entities`, `classes`, and `columns` represent different possibilities of mapping. It's important that only one specification is possible, not several.

❶ With `entities`, the columns are transferred to an entity bean.

❷ `constructorResult` specifies a parameterized constructor, and the Jakarta Persistence provider constructs the result object using this constructor. (We'll use this in a moment.)

❸ With `columnResult`, the Jakarta Persistence provider builds an uninitialized object via the *parameterless* constructor and will subsequently populate the corresponding values via setters. In this case, a mapping must be created between the columns and the target object's properties.

@SqlResultSetMapping with @ConstructorResult

The following code snippet summarizes the native query with `@NamedNativeQuery` and `@SqlResultSetMapping`:

```
@NamedNativeQuery(
    name = "Profile.containsTwoVowelsNickname",
    query = """
            SELECT nickname, manelength
            FROM profile
            WHERE REGEXP_LIKE(nickname, '[aeiou]{2}', 'i')""",
    resultSetMapping = "nicknameManelengthMappingToProfileValueObject" )

@SqlResultSetMapping(
    name = "nicknameManelengthMappingToProfileValueObject",
    classes = {
        @ConstructorResult(
            targetClass = ProfileValueObject.class,
            columns = {
                @ColumnResult( name = "nickname" ),
                @ColumnResult( name = "manelength" )
            }
        )
    }
)
```

We know the first part, but what is new in `@SqlResultSetMapping` is the assignment of `classes`, an array in which we specify only one element. `@ConstructorResult` determines with `targetClass` the name of the class or record from which the Jakarta Persistence provider should later call the parameterized constructor. In our case, this is `ProfileValueObject`.

The next step is to map the columns to the corresponding parameters of the parameterized constructor. A parameterized constructor has parameters at position 0, 1, 2, and so on, and they are enumerated in order at `@ColumnResult`. Therefore, `@ColumnResult` has the column name (or alias) `nickname` first because that is the first parameter in the constructor. The mane length follows it.

In this way, columns of the result can be mapped to a separate object. Basically, we've described the ORM manually. The `EntityManager` doesn't have to manage the objects.

Calling the Native Query

The call of the native query succeeds again with `createNamedQuery(…)`:

```
em.createNamedQuery( "Profile.containsTwoVowelsNickname",
                     ProfileValueObject.class )
  .getResultList()
  .forEach( simpleProfile -> log.info( "{}", simpleProfile ) );
```

The logically chosen name `Profile.containsTwoVowelsNickname` and the type token `ProfileValueObject` are passed. Then, the compiler correctly recognizes that the list contains `ProfileValueObject`. With `createNativeQuery(…)`, there was no type information.

6.8 Write Access with the EntityManager in Transactions

Until now, we've only retrieved data from entity beans using the `EntityManager`. We've used methods such as `find(…)` and `getReference(…)`, as well as read JPQL statements. However, modifying data—that is, storing, updating, and deleting it—requires a special approach.

6.8.1 Transactional Operations

`EntityManager` provides several methods for writing and modifying:

- `persist(…)`
- `remove(…)`
- `merge(…)`
- `refresh(…)`
- `flush()`
- JPQL with change and execution via `executeUpdate()`

An important requirement for use is that all changes to the database must happen in transactions. Without a transaction, a call to the preceding methods will result in a `TransactionRequiredException`.

6.8.2 persist(…)

`EntityManager` allows us to not only read but also modify data by storing, updating, and deleting objects. To store a new object, we use the `void persist(Object entity)` method, which places it in the persistence context. This method takes an argument of type `Object`, representing the new entity bean to be stored, and has no return. Typically, the primary key for the entity is set automatically by the database, for example, through an identity or sequence column, and the `persist(…)` method will update the key column accordingly. In our case, if we were to save a profile, the generated key would be assigned to the instance variable ID by the Jakarta Persistence provider.

However, errors can occur with the `persist(…)` method. For example, if an ID is mistakenly set for the entity, but the database assigns a different one, an error will occur during saving. We'll explore the concept in Section 6.9.

[!]

Warning: Important

The `persist(…)` method may be used only when this entity is newly created. For updating entities, `persist(…)` must not be used. If there is already an entity with the ID, an `EntityExistsException` follows.

In a transaction, `persist(…)` will store the object. But how do we mark a transactional block? There are two ways to do this.

6.8.3 EntityTransaction

For manual control of a transaction, an `EntityTransaction` object can be requested from `EntityManager`. This `EntityTransaction` object is always connected to the `EntityManager`; there is a 1:1 relationship between the two.

The `EntityTransaction` interface declares the following methods:

- `void begin()`
  Starts a transaction.
- `void commit()`
  Completes the transaction and writes data that is in the buffer to the database.
- `void rollback()`
  Undoes the operations of the transaction.
- `boolean isActive()`
  Checks if a transaction is currently running.
- `void setRollbackOnly()`
  Processing isn't directly aborted and continued, but the transaction isn't committed at the end.
- `boolean getRollbackOnly()`
  Queries the rollback-only flag.

The most important methods are `begin()`, `commit()`, and `rollback()`.

Programmed Transactional Bracket

In practice, this could be used as follows:

```
EntityTransaction tx = em.getTransaction();
tx.begin();
…
tx.commit();
```

Called on the `EntityManager`, `getTransaction()` returns the `EntityTransaction` object. `begin()` is used to start the transaction. At the end, `commit()` is called. Because the `EntityTransaction` object is exclusively associated with the `EntityManager`, we can always receive this object with `getTransaction()`, and then we can get by without an intermediate variable:

```
em.getTransaction().begin();
…
em.getTransaction().commit();
```

The small program fragment doesn't demonstrate exception handling. If you choose to use the programmed approach, a `try-catch` block can be used to catch exceptions and reset the transaction with the `rollback()` method in the event of an error.

Although it's technically possible to include exception handling directly in the program code, it's not the recommended approach in practice, especially in Spring. The preferred method is declarative usage, which was discussed in Chapter 5, Section 5.11.6.

6.8.4 PlatformTransactionManager with JpaTransactionManager

The `EntityTransaction` data type is part of Jakarta Persistence. However, Spring has its transaction manager type called `PlatformTransactionManager`. This type serves as an abstraction for all transactions implemented in a Spring context, whether they are for relational databases or messaging systems. Therefore, it's not recommended to use `EntityTransaction` directly in Spring applications. Instead, a Spring program should exclusively use `PlatformTransactionManager`, which can work in the background with `JpaTransactionManager` (which uses `EntityTransaction`).

The UML diagram in Figure 6.7 shows that the `org.springframework.transaction` package contains the general type `PlatformTransactionManager`. This interface has several implementations, including ones for JDBC, which are used for relational database management systems. If working with Jakarta Persistence, Spring will automatically register a `JpaTransactionManager`, which internally accesses the `EntityTransaction` of the `EntityManager`.

That is, a Spring program will always work with Spring's transactional infrastructure, just as we saw before. This is done either with the methods of `PlatformTransactionManager` or with annotation `@Transactional`.

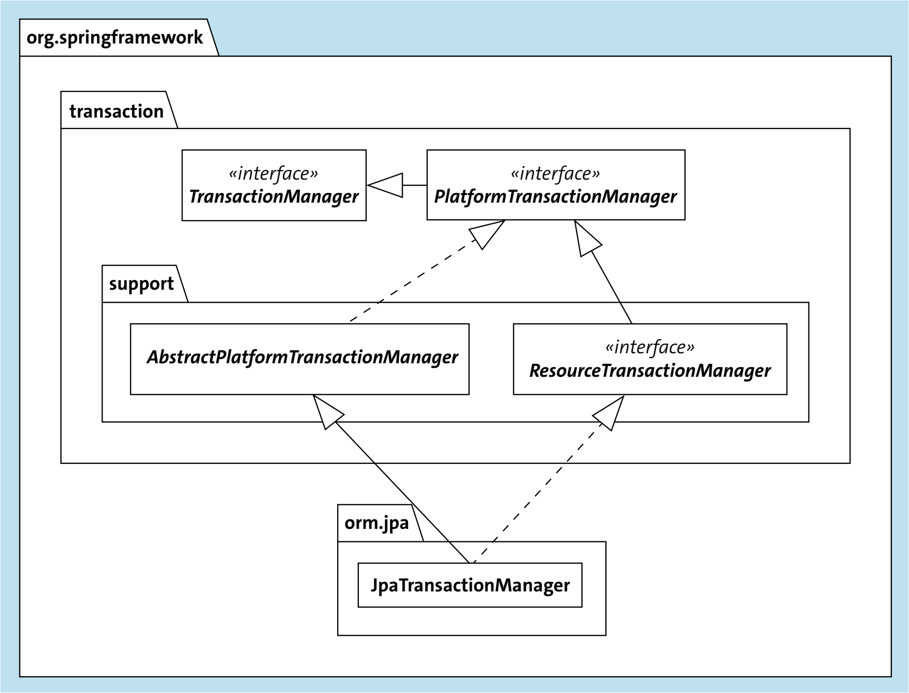

org.springframework
transaction
«interface»
TransactionManager
«interface»
PlatformTransactionManager
support
AbstractPlatformTransactionManager
«interface»
ResourceTransactionManager
orm.jpa
JpaTransactionManager

Figure 6.7 “JpaTransactionManager”: A “PlatformTransactionManager”

JpaTransactionManager

For a Jakarta Persistence API application, Spring Boot initializes a concrete `JpaTransactionManager` for `PlatformTransactionManager` through auto-configuration. You can think of it like this:

```
@Bean
PlatformTransactionManager transactionManager( EntityManagerFactory emf ) {
  return new JpaTransactionManager( entityManagerFactory );
}
```

A program can inject `PlatformTransactionManager` and perform the programmed transaction management using `commit(…)` and `rollback(…)`.

However, it’s easier to work declaratively with `@Transactional`, as we’ll show next.

6.8.5 @Transactional

With annotation `@Transactional` (no matter if from package `jakarta.transaction` or `org.springframework.transaction.annotation`) at the type or at the (`public`) method, you switch on the transactional control. Then, you can use modifying operations such as `persist(…)`, `remove(…)`, and so on in the block.

Here are two important notes to consider:

- Transactional methods must be `public` so that a corresponding transactional proxy can access the methods. Other visibilities are possible in principle, but the bytecode may have to be modified via AspectJ.
- It's necessary for methods to always access the transactional core from the outside via the proxy. If a method within the core calls another method within the core, the transactional proxy shouldn't be bypassed. If a nontransactional method within the same class calls an `@Transactional` method, no transactions will be initiated or continued, which is considered an error.

persist (…) in a @Transactional Method

Here's an example that demonstrates the usage of the `persist(…)` method:

```
@Transactional
public void run() {
  EntityBean newEntityBean = new EntityBean();
  newEntityBean.set*( * );
  newEntityBean.set*( * );
  newEntityBean.set*( * );
  newEntityBean.set*( * );
  em.persist( newEntityBean );
}
```

The first thing that stands out is `@Transactional`. If this annotation is missing, no transaction will be executed, and the result will be a `TransactionRequiredException`, unless the method is called from elsewhere, and the transaction has already been started.

Depending on whether we have a parameterless or a parameterized constructor, a new entity bean instance would be constructed for saving. For example, if the symbolic class `EntityBean` has a parameterless constructor, the object will be constructed and later filled via the setters.

If the entity bean is immutable, as in the case of our `Profile` class, no setters are called and the parameterized constructor accepts all data and saves it. At the end, the `persist(…)` method writes the row.

6.8.6 Save versus Update

In SQL, there is a difference between "store new" and "store old," as it distinguishes between `INSERT` (for saving new data) and `UPDATE` (for updating existing data). Although in Spring Data, there is a single `save(…)` method, the `EntityManager` makes a distinction and provides two methods: `persist(…)` and `merge(…)`. The `persist(…)` method should exclusively be used for saving new rows, while `merge(…)` is used for updating existing ones. Entities are always considered new if they don't yet exist in the database under the respective ID. If the database generates the ID, the persistent attribute of the ID must be `null` when using `persist(…)`.

6.8.7 remove(…)

The `remove(…)` method deletes an entity bean. The object must have been loaded beforehand; that is, it must be in the persistence context (more about this in Section 6.9).

Here's an example: method `remove(…)` should delete the persistent object with ID `1` from the database:

```
Profile fillmore = em.find( Profile.class, 1L );
em.remove( fillmore );
```

However, the operation fails because the database prevents deletion: there are other records attached to the profile, such as photos, which must be deleted first.

`remove(…)` can result in a cascade of further delete operations. In our example, this could mean that when a profile is deleted, the attached photos are also deleted. Whether something should be deleted in a cascade can be determined declaratively (we'll discuss this topic in Section 6.13).

6.8.8 Synchronization or Flush

`EntityManager` is responsible for synchronizing the states of entity beans in main memory with the tables in the database by executing SQL statements at certain times. This process is called *flush*. The timing of this synchronization, which is implementation-dependent, is determined by the Jakarta Persistence provider. For example, Hibernate performs synchronization at the following points:

- Before each request, when used by `query`
- On a `commit()` of `EntityTransaction`
- For a `flush()` from `EntityManager`

By calling the `flush()` method, a program can deliberately synchronize changed entity beans with the database. However, an active transaction is necessary to call `flush()`; without it, the result will be a `TransactionRequiredException`.

flush() or refresh(…) for SQL Triggers

The `flush()` method isn't commonly used, but it can be valuable when database triggers modify the row after our write. Then the persistence provider isn't aware of it, and the data in memory becomes outdated. `EntityManager` can refresh the data using the `refresh(…)` method.

Unlike the `find(…)` method, which requires an ID as an argument, the `refresh(…)` method needs the complete entity bean object. The `refresh(…)` method then refills the entity bean object with the latest data from the database. In this manner, if triggers have modified the entities, these changes can be read in directly again.

6.8.9 Query with UPDATE and DELETE

JPQL isn't a pure query language, but it can also update and delete. This is intended for large mass data in *batch updates*.

Here we see such an example of a modification:

```
em.createQuery( "DELETE FROM Photo p" ).executeUpdate();
```

This will delete all photos from the database.

In addition to `DELETE`, `UPDATE` can also be used. Changes to the database are only possible in one transaction.

Tip

These operations are designed for batch updates. JPQL is converted to SQL, allowing for the fast modification or deletion of numerous rows. Loading, modifying, and saving a new entity bean in a loop shouldn't be performed by a program as it can be slow and usually can't be optimized by the Jakarta Persistence provider. Conversely, if a program wants to delete a single row, it's typically more straightforward and readable to use the `remove(…)` method instead of issuing a `DELETE` statement in a JPQL expression with an ID.

6.9 Persistence Context and Other Transaction Controls

Persistence context has already been mentioned a few times, but its meaning and exact function should now be clarified in detail. `EntityManager` is always connected to a persistent context, and this is comparable to a persistent data structure that writes changed rows in the background. Of course, the persistence context doesn't manage arbitrary objects as a data structure does, but only entity beans that are connected to a database.

When using Jakarta Persistence *outside* of Spring, *persistent unit files* are required, in which database connections, entity beans, and metadata are listed.

The persistence context isn't controlled directly but indirectly via the methods of `EntityManager`. For example, when we add an object to `EntityManager` and persist it, this object is *managed*. Each managed entity bean can be transferred to other states.

6.9.1 Jakarta Persistence API and Database Operations

Until now, it looked like there was a direct interaction between the Jakarta Persistence API and database operations with SQL:

- `find(…)` → `SELECT`
- `persist(…)` → `INSERT`

- `remove(…)` → `DELETE`
- `merge(…)` → `UPDATE`

Although it may appear that a method calls results in a direct database operation, this isn't entirely accurate in the case of Jakarta Persistence. While some O/R mappers may operate in this way, Jakarta Persistence is designed differently. Objects are managed in the persistence context, and when the transaction permits, an exchange between the persistence context and the database takes place. In other words, SQL statements are indirectly triggered by changes in the persistence context, rather than directly through the methods.

6.9.2 States of an Entity Bean

An entity bean can be in different states. The moment you create a new object, it has nothing to do with the persistence context. The state is called *new*. If an object has a database identity and is connected to a persistence context, then the object is called *managed*. Objects can also be kicked out of the persistence context. These objects are then called *detached*. Detached objects have a database identity but aren't currently in the persistence context. The *removed* state means that the entity bean originally had a database identity, but is about to be deleted and will also disappear from the persistence context.

State Transitions

An entity bean can change from one state to another. The best way to see which methods lead to which states is by looking at a state diagram (see Figure 6.8).

One way to create an object is by using the `new` keyword, which creates an object in a *transient* state that isn't yet persistent. In other words, the object has no database identity and isn't part of the persistence context. To make the object persistent, the `persist(…)` method is called, which changes the object's state to managed and triggers an SQL `INSERT`. However, whether the SQL statement is executed directly or not depends on the Jakarta Persistence provider implementation. Normally, synchronization with the database occurs when the transaction is closed or when a flush is performed, but `persist(…)` only changes the state of the object to managed within the `EntityManager`. The `EntityManager` can be thought of as a collection of entities that are managed, and a state change of the entity bean leads to synchronization with the database.

Another way to bring objects into the managed state is to load them using the `find(…)`, `getReference(…)`, or `get*()` method on a `[Typed]Query`. The managed state can be transitioned to the removed state by calling the `remove(…)` method, which ultimately results in a `DELETE` operation on the database side. If an object is removed, its ID becomes invalid, but it can be brought back to the managed state using the `persist(…)` method. This triggers another `INSERT`, as long as the ID is reset when generating the database.

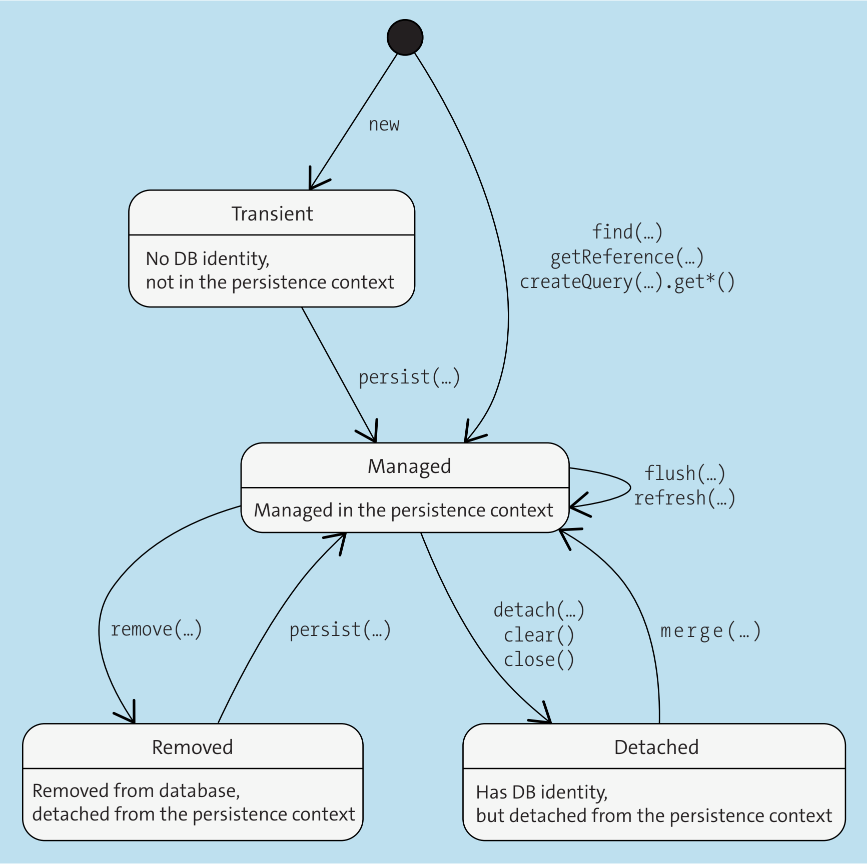

new
Transient
No DB identity,
not in the persistence context
find(…)
getReference(…)
createQuery(…).get*()
persist(…)
Managed
Managed in the persistence context
flush(…)
refresh(…)
remove(…)
persist(…)
detach(…)
clear()
close()
merge(…)
Removed
Removed from database,
detached from the persistence context
Detached
Has DB identity,
but detached from the persistence context

Figure 6.8 State Transitions of an Entity Bean

Several methods can detach a managed object, that is, remove it from the persistence context. The `detach(…)` method removes a single object from the managed state, `clear()` removes all entities from `EntityManager`, and `close()` closes `EntityManager`, releasing all entity beans. Although the data still exists on the database side, it's no longer part of the `EntityManager`'s data structure. However, because they have an identity, `merge(…)` can be used to make them a managed entity bean again.

State Changes Detected in Transactions

Let's look at the following example:

```
Profile fillmore = em.find( Profile.class, 1L );
log.info( "{}", fillmore.getManelength() );
fillmore.setManelength( fillmore.getManelength() + 1 );
log.info( "{}", fillmore.getManelength() );
log.info( "{}", em.find( Profile.class, 1L ).getManelength() );
```

The process involves loading the profile with ID 1 and logging its mane length, followed by incrementing the length by one using the setter and logging the new value. Thereafter, the profile is reloaded, and the old value, followed by the old value plus one, is logged again.

While this sequence of events isn't particularly noteworthy, things change when transactions come into play. If `setManelength(…)` were enclosed in a transactional block, the profile would be updated and the Jakarta Persistence provider would generate a SQL `UPDATE`. A transactional block would be initiated, for instance, if the method's top rows were annotated with `@Transactional`.

Automatic Dirty Checking

The detection of changes in a transaction is called *automatic dirty checking*. This is a feature that every persistence provider must realize. A call to `merge(…)` to update is no longer necessary at all because everything that has been changed in a transaction is written automatically. This is really distinctive and, to a certain extent, magical.

It's not unusual to find remnants in the code that indicate a database update. This isn't necessarily incorrect, especially if there are plans to migrate from Spring Data JPA to Spring Data JDBC, as the latter doesn't have automatic dirty checking and requires explicit saving.

Automatic dirty checking only works if the object is in the persistence context and, therefore, managed. With a closed `EnitiyManger` or a detached object that isn't in context, no changes are detected.

Moving on to the technical implementation, Hibernate provides two different strategies:

- The original implementation is called *diff-based* or *snapshot*. It stores component states. When synchronization with the database occurs, Hibernate compares the old states with the new state, detects changes, and stores the modified entities. This is resource-intensive because it requires the entity's memory footprint times two.
- A more modern approach to Hibernate generates bytecode on all state modifications, so that Hibernate is aware of where a write operation changes the state of the entity. This is a high-performance and memory-saving option. However, mutable objects, such as `Date` objects, are problematic because you can't just see that the reference has changed; you have to look inside the container as well. But if it's possible to work via *bytecode-enhanced dirty tracking*, then the program can save memory. The techniques are quite interesting, and the Hibernate User Guide at *https://docs.jboss.org/hibernate/orm/current/userguide/html_single/Hibernate_User_Guide.html* has both suggestions on how to enable this option and information on possible pitfalls.

Task: Test Entity Monitoring

We've noted that Hibernate can automatically detect state changes via dirty checking and write away the changes during a database synchronization. A new method of our own is intended to show this feature.

We added a new method to `EntityManagerCommands` for an `update-manelength` command that changes the mane length of a profile. The ID of the profile and the new mane length will be passed. We have to make sure that the shell method is `public`. Then, we test the command first without `@Transactional` (nothing will happen) and then with `@Transactional` (saving is done). We don't use `merge(…)` because it's not necessary.

Proposed solution: The new method will be called `updateManelength` and will take a `long` for the `id` and `int` for the mane length. The profile will be loaded to get it into the persistence context, and then changed. If the method is written without `@Transactional`, nothing will happen.

```
@ShellMethod( "Set mane length of a given profile" )
public void updateManelength( long id, int manelength ) {
  Optional.ofNullable( em.find( Profile.class, id ) )
          .ifPresent( p -> p.setManelength( manelength ) );
}
```

Listing 6.8 EntityManagerCommands.java Extension

Only when the public method is annotated with `@Transactional`, the transactional proxy wraps around the method and opens the transactional bracket, which activates Hibernate's dirty checking. The setter triggers an SQL `UPDATE` due to the modification.

```
@Transactional
@ShellMethod( "Set mane length of a given profile" )
public void updateManelength( long id, int manelength ) { … }
```

Listing 6.9 EntityManagerCommands.java Extension

6.10 Advanced ORM Metadata

We've already discussed the various Jakarta Persistence annotations. In this section, we build on that knowledge.

6.10.1 Database-First and Code-First Approaches

For annotations, we distinguish between *physical mapping information* (e.g., the name of the table or column on one side) and *logical mapping information* (e.g., the fact that a column contains an ID or a persistent attribute represents a 1:1 relationship). Jakarta Persistence doesn't work without this mapping information, and annotations or XML files can be used for the description. Because both are equivalent, we'll continue to use only annotations for the description.

For Jakarta Persistence to work properly, the type and mapping information on the Java side must match that of the database tables. As we discussed in Section 6.1.2, this can be a challenge due to the "impedance mismatch" between OO models and database models. However, there are two ways to approach the development process: database-first and code-first.

- **Database-first approach**
  In this approach, a preexisting database schema exists, and the corresponding entity beans must be formed from it. This is the more common approach, and, fortunately, there are tools available that can connect to a database and generate entity beans automatically. However, the quality of the resulting entity beans can vary depending on the tool used.
- **Code-first approach**
  In this approach, the fully annotated entity beans are created first, and a database is then generated based on these beans. This approach requires specifying as much metadata as possible on the Java side to ensure reliable database generation. The Jakarta Persistence provider is a tool that can automatically create the necessary database tables and relationships based on the entity beans.

6.10.2 Set the Table Name via @Table

The Jakarta Persistence provider assumes by default that the table on the database side has the same name as the entity bean class. One advantage is that the case isn't relevant. However, if the name of the table is different from the entity bean class name, the table name can be explicitly set using `@Table`.[19] Here's an example:

```
@Entity
@Table( name = "unicornprofiles" )
public class Profile
```

Thus, class names and table names are decoupled from each other.

Setting the name can be useful or even necessary for the following reasons:

- If the table name contains spaces or other invalid Java characters
- To comply with naming conventions: no underscores in type names, everything in English, singular/plural renaming, and so on
- If the table name conflicts with a Java data type, such as `Class`

Besides `name`, `@Table` has other annotation attributes: `catalog`, `schema`, `indexes`, and `uniqueConstraints`.

---

[19] *https://jakarta.ee/specifications/persistence/3.1/apidocs/jakarta.persistence/jakarta/persistence/table*

View

The entity bean doesn't necessarily correspond to a physical table, but it can also be associated with a database view. In this case, the name of the view should be specified in the `@Table` annotation. The Jakarta Persistence provider isn't concerned with whether the entity bean maps to a physical table or a view. A view can offer an additional level of security, for instance, by restricting write access while allowing read access. However, it's essential to be aware of the type of changes that can be made to views in certain database management systems.

6.10.3 Change the @Entity Name

Besides setting the table name explicitly so that the class can be named differently from the table, we can also change the entity name for JPQL expressions. This is necessary, for example, if the table name collides with a keyword from JPQL. For instance, if we have a table called `Order` for an order—but `ORDER` is also a JPQL keyword for sorting for an order via `ORDER BY`. If the table is called `Order`, the entity class can also be called `Order`, but a different entity name makes sense.

The entity name is set at `@Entity`.[20] The annotation type is simple with only one attribute for the name:

```
@Documented
@Target(TYPE)
@Retention(RUNTIME)
public @interface Entity {
        String name() default "";
}
```

It's important to understand the difference: The entity name is used exclusively in JPQL, while the table name is a property of the physical mapping.

6.10.4 Persistent Attributes

The Jakarta Persistence provider must take the data from the entity bean and write it back. The standard defines two different strategies:

- **Persistent instance variables (persistent fields) and field-based access**
  The provider uses reflection to set and read entity bean instance variables. The annotations, such as `@Id`, are applied to the instance variables.
- **Persistent properties and property-based access**
  The Jakarta Persistence provider calls setter/getter to indirectly set and read the entity instance variables. Annotations are placed at the JavaBean property methods. It doesn't matter whether they are at the setter or getter.

---

[20] *https://jakarta.ee/specifications/persistence/3.1/apidocs/jakarta.persistence/jakarta/persistence/entity*

Persistent instance variables and persistent properties are generally referred to as *persistent attributes*—this is also what the specification does.

Determine the Access Type

Whether the persistence provider uses the instance variables or setter/getter is determined via annotation `@Access`.[21] For this purpose, there is enumeration `AccessType`[22] with two constants: `AccessType.FIELD` and `AccessType.PROPERTY`. We had already set it for `Profile`.

```
@Entity
@Access( AccessType.FIELD )
class …
```

Listing 6.10 Profile.java

Note

The Jakarta Persistence specification doesn't provide a default setting for `@Access`. This means that it's not specified whether `AccessType.FIELD` or `AccessType.PROPERTY` should be used. However, Hibernate does have a default strategy, which is documented at *https://docs.jboss.org/hibernate/orm/current/userguide/html_single/Hibernate_User_Guide.html#access*. According to the documentation:

> *By default, the placement of the @Id annotation gives the default access strategy. When placed on a field, Hibernate will assume field-based access. When placed on the identifier getter, Hibernate will use property-based access.*

In the case of our entity bean profile, we don't need to set `AccessType.FIELD` because we've set `@Id` on the instance variable. However, it's still recommended to explicitly set `AccessType.FIELD`.

Rules for Persistent Attributes

Some rules apply to persistent attributes. According to the Jakarta Persistence standard, persistent attributes must not be `final`. This is obvious because if an entity bean is filled, then the persistence provider must be able to write this variable. With `AccessType.PROPERTY`, there will be setters, and they must also be able to write the internal variables. The standard further says that the persistent instance variables shouldn't be `public`.

---

[21] *https://jakarta.ee/specifications/persistence/3.1/apidocs/jakarta.persistence/jakarta/persistence/access*

[22] *https://jakarta.ee/specifications/persistence/3.1/apidocs/jakarta.persistence/jakarta/persistence/accesstype*

Immutable entity beans are possible in principle, if we write a `protected` constructor and offer a public parameterized constructor for us and only getters. Without setters, the variables can then never be updated from the outside.

6.10.5 @Basic and @Transient

Depending on the `AccessType` set, all nonstatic persistent attributes are automatically persistent by default. To omit a persistent attribute, set annotation `@Transient`.[23] This may be necessary if the entity bean has other internal states, such as cached values, that have nothing to do with a persistent attribute,

In addition to `@Transient`, annotation `@Basic`[24] explicitly identifies a persistent attribute, which might increase readability. This is actually unnecessary because all persistent attributes are considered anyway. We therefore omit `@Basic`, although the annotation may be useful as an anchor for the fetch type to load the persistent attributes `LAZY` and `EAGER`; we'll discuss this later in Section 6.12. Moreover, via `@Basic`, the annotation attribute can be optionally set, which determines whether the column can be `null`. But this is better done via `@Colum`.

6.10.6 Column Description and @Column

The `@Column`[25] annotation is used to specify various properties of database columns, such as whether they can be `NULL`, the number of decimal places for floating-point numbers, and the column width for strings. This information can be described completely on the Java side using the `@Column` annotation, which is declared as follows:

```
@Target({METHOD, FIELD})
@Retention(RUNTIME)
public @interface Column {
 // Column name (By default, this is the name of the property)
 String name() default "";
 // The column's unique constraint
 boolean unique() default false;
 // The column can be NULL
 boolean nullable() default true;
 // The column is part of the INSERT/UPDATE.
 // If false, it will be ignored by the provider
 boolean insertable() default true;
```

---

[23] *https://jakarta.ee/specifications/persistence/3.1/apidocs/jakarta.persistence/jakarta/persistence/transient*

[24] *https://jakarta.ee/specifications/persistence/3.1/apidocs/jakarta.persistence/jakarta/persistence/basic*

[25] *https://jakarta.ee/specifications/persistence/3.1/apidocs/jakarta.persistence/jakarta/persistence/column*

```
 boolean updatable() default true;
 // Overwrites the SQL DDL fragment
 String columnDefinition() default "";
 // Target table (default is the current table)
 String table() default "";
 // Column width
 int length() default 255;
 // Decimal precision int precision()
 default 0;
 // Decimal places
 int scale() default 0;
}
```

Understanding the relationship between the `@Column` annotation and the database schema is crucial. The database always has a schema with a complete description. The `@Column` annotation is particularly useful when working with the code-first approach and a schema is created by the Jakarta Persistence provider. This requires maximum information for high-quality database generation.

Some metadata is only relevant during generation, while other metadata, such as the column name, is essential for proper O/R mapping functionality. Some information can't even be set via standard Jakarta Persistence, and proprietary solutions of the providers are necessary; therefore, Hibernate offers, among others, the `@Check` annotation.[26]

Naming Strategy

Similar to how the Jakarta Persistence provider derives the name of a table from the name of the entity bean class by default, it also assumes that the name of a persistent attribute corresponds to the name of a column in the database. However, this default behavior can be modified or manually assigned using the `@Column` annotation, as we did in the case of a column in the `Photo` entity.

```
@Entity
@Access( AccessType.FIELD )
public class Photo {
  …
  @Column( name = "is_profile_photo" )
  private boolean isProfilePhoto;
  …
}
```

Listing 6.11 Photo.java

---

[26] *https://docs.jboss.org/hibernate/orm/current/javadocs/org/hibernate/annotations/Check.html*

Persistent attribute `isProfilePhoto` is set to column name `isprofilephoto` by default according to the Jakarta Persistence standard. Because the column name can have underscores on the database side, `@Column` corrects the name so that a known naming convention with camel case is followed on the Java side.

Essentially, we have a conversion of naming conventions from camel case on the Java side to snake case with underscores on the database side. While the `@Column` annotation can be used to customize column names, there is another option available called a column naming strategy. With this strategy, the persistence provider can automatically modify naming conventions, such as splitting camel case attributes into underscores or minuses, or removing separators altogether. In Spring Boot, Hibernate's `CamelCaseToUnderscoresNamingStrategy` is set by default, so Spring Data JPA ignores the mapping from the Jakarta Persistence specification and introduces its own naming strategy. Therefore, in our case, using the `@Column` annotation with the name parameter (e.g., `@Column(name = "is_profile_photo")`) isn't even necessary when developing a Spring application.

Setting the annotation for the column name has one advantage: if the entity bean is ported outside of a Spring application, for example, in a typical Jakarta EE application, and if this application respects the naming from the Jakarta Persistence standard, then it would fail without `@Column`.

Setting the Length of a Text Column at @Column

With the `@Column` annotation, the length of a text column can also be specified. In this example, a text column is to be set to 2048 characters.

```
@Column( length = 2048 )
String description;
```

Listing 6.12 Profile.java Extension

This information isn't important if the table has already been created. However, if a table is generated from the entity bean metadata, the information is extremely valuable because, by default, these columns would otherwise be only 255 characters wide.

Note

When using a Jakarta Persistence provider, the metadata contained in the `@Column` annotation isn't used for validation purposes. Therefore, if the text column contains more text than its defined limit, the Jakarta Persistence provider will truncate the text without raising any validation errors during the saving process. Because strings in Java can be of any length, but RDBMS columns for strings have a maximum length, errors of this nature can be caught by using Jakarta Bean Validation.

6.10.7 Entity Bean Data Types

The stored states of an entity bean can be roughly divided into two categories:

- *Basic attributes* are values that can be written directly in a column. These include numeric values or character strings.
- Conversely, an entity bean may contain references to other entity beans. This is where a Jakarta Persistence provider can be particularly advantageous, as it can automatically resolve 1:1 or 1:n relationships.

Supported Data Types for Basic Attributes

An entity bean can use the following data types for basic attributes:

- All primitive Java types and their wrapper classes
- `String`
- `BigInteger`, `BigDecimal`
- `LocalDate`, `LocalTime`, `LocalDateTime`, `OffsetTime`, `OffsetDateTime`, as well as `java.util.Date`, `Calendar`, `java.sql.Date`, `Time`, and `Timestamp`
- `Enumerations` (`enum`) (into an integer or string column)
- `byte[]`, `Byte[]`, `char[]`, `Character[]`
- Serializable objects (into a `LOB`)

If the basic attribute types fall short, it's possible to develop converters. These can be employed to present custom data types in textual format, among other things.

Note

As with strings, a program must consider that a Java data type may carry more information than the database type and that data loss will occur. Numbers in `BigInteger` can be of any size, and `BigDecimal` even has an arbitrary number of decimal places. A database usually can't handle that, so you have to be careful here that there is no data loss. This is where the Jakarta Bean Validation can help.

Temporal Types

Jakarta Persistence supports the "modern" types of the Date-Time API: `LocalDate`, `LocalTime`, `LocalDateTime`, `OffsetTime`, and `OffsetDateTime`. The Java types can be mapped to database columns of the types `DATE`, `TIME`, or `TIMESTAMP`. Special annotations aren't necessary for the mapping.

In older Java programs, data types `java.util.Date` and `java.util.Calendar` are occasionally used. These can also be mapped through Jakarta Persistence, but there is a problem: `Date` and `Calendar` have a date and time component, so annotation `@Temporal` is required, for example, like this:

```
@Temporal( TemporalType.DATE )
java.util.Date getBirthday() { … }
```

Annotation `@Temporal`[27] determines the precision of the Java type (time, date, or both).

Enumeration (Java Enum) and @Enumerated

Jakarta Persistence can also map enumeration types. Suppose an enumeration type is declared as follows:

```
enum Gender { FEE, MAA }
```

Then, an entity bean could have a persistent `Gender gender` attribute, and the Jakarta Persistence provider could easily map this.

The question is how to map this reference to a database column. Jakarta Persistence has two strategies that are set via `@Enumerated`:[28]

- When using `@Enumerated(EnumType.ORDINAL)` (which is the default), the integer column will contain the ordinal value (provided by the `ordinal()` method) of the enum. The ordinal number corresponds to the position of the element in the `enum`. For example, `FEE` would have an ordinal value of `0`, and `MAA` would have an ordinal value of `1`.
- When using `@Enumerated(EnumType.STRING)`, the string column will contain the string representation of the `enum` (provided by the `name()` method). Therefore, in this case, the values in the column would be the names of the `enum` elements (i.e., `FEE` and `MAA`).

Here's an example in which the `enum` is to be written as a string:

```
@Enumerated( EnumType.STRING )
Gender gender;
```

Both options have drawbacks. If the variable name changes, it becomes impossible to retrieve data from the database because the name is no longer valid. Similarly, the ordinal method only works well when new constants are added at the end; if constants are reordered, the assignment between the ordinal number and the constant becomes incorrect.

The concerns raised highlight the challenges of this method, suggesting the need for alternative solutions. One such solution is the use of attribute converters, which we'll discuss in Section 6.10.8. With an attribute converter, `FEE` could be mapped to `1` and `MAA` to `2`, decoupling the enumeration's semantics from the name or ordinal number.

---

[27] *https://jakarta.ee/specifications/persistence/3.1/apidocs/jakarta.persistence/jakarta/persistence/temporal*

[28] *https://jakarta.ee/specifications/persistence/3.1/apidocs/jakarta.persistence/jakarta/persistence/enumerated*

Binary Large Objects or Character Large Objects with @Lob

Huge objects can be stored in a database using the data types CLOB (*character large object*) or BLOB (*binary large object*).

Jakarta Persistence maps the binary or text columns to Java data types via annotation `@Lob`[29] to Java data types:

- **CLOB**
  `char[]`, `Character[]`, `java.lang.String`, and `java.sql.Clob`
- **BLOB**
  `byte[]`, `byte[]`, serializable types, and `java.sql.Blob`

The annotation type `@Lob` has no attributes.

Suppose an XML document is stored in a CLOB, and an image is stored in a BLOB on the database side. In that scenario, we can accomplish the necessary retrieval and storage operations using the `@Lob` annotation:

```
@Lob private String xml;
@Lob private byte[] image;
```

6.10.8 Map Data Types with AttributeConverter

Jakarta Persistence maps various data types to the database, but the number is limited. For example, for `Locale`, `Currency`, `Path`, and `DataSize`, the standard doesn't define any mappings. In addition, the defined mappings may not be optimal. For example, a string may need to be mapped to an encoded string, or, in the case of `enum`, a custom mapping would be useful.

`jakarta.persistence.AttributeConverter`[30] can be used to map any Java type to an existing database type. `AttributeConverter` is an interface with two methods:

```
package jakarta.persistence;

public interface AttributeConverter<X,Y> {
  Y convertToDatabaseColumn( X attribute );
  X convertToEntityAttribute( Y dbData );
}
```

These methods convert between the Java type on one side and the database type on the other side and back again.

---

[29] *https://jakarta.ee/specifications/persistence/3.1/apidocs/jakarta.persistence/jakarta/ persistence/lob*

[30] *https://jakarta.ee/specifications/persistence/3.1/apidocs/jakarta.persistence/jakarta/persistence/attributeconverter*

AttributeConverter Example

Suppose that we need to write a list of strings from the Java side as a comma-separated string on the database side. The following implementation of the `AttributeConverter` interface achieves this:

```
@Converter
public class CommaSeparatedStringToListConverter
    implements AttributeConverter<List<String>, String> {

  @Override
  public String convertToDatabaseColumn( List<String> attribute ) {
    return attribute == null || attribute.isEmpty()
           ? ""
           : String.join( ",", attribute );
  }

  @Override
  public List<String> convertToEntityAttribute( String column ) {
    if ( column == null || column.isBlank() )
      return new ArrayList<>();

    return new ArrayList<>( Arrays.asList( column.split( "," ) ) );
  }
}
```

Two types of information must be specified during the implementation: the Java source type (a list of strings) and the Java type to be used on the database side (`String`). This can be mapped via the default mapping to, for example, a `VARCHAR`.

The first method, `convertToDatabaseColumn(…)`, gets a list of strings, and these are converted to a comma-separated string. The opposite method, `convertToEntityAttribute(…)`, gets the column contents from the database and converts them to a list. The implementation returns all returns as a modified list.

The `AttributeConverter` must be registered in the next step.

Apply AttributeConverter

Following are the various options available for registering converters, but we won't consider the XML variant for now:

- Global: Set `autoApply=true` on the converter class:

  ```
  @Converter( autoApply = true )
   public class CommaSeparatedStringToListConverter
      implements AttributeConverter<List<String>, String>
  ```

  This applies the converter globally to all lists of strings.

- Add to the persistent attribute `@Convert(converter = …)`:

```
@Convert( converter = CommaSeparatedStringToListConverter.class )
 private List<String> strings = new ArrayList<>();
```

Warning

The annotation is called `@Convert`, not `@Converter`!

- At the entity bean class:

```
@Entity
@Converts( {
  @Convert( attributeName = "strings",
            converter = CommaSeparatedStringToListConverter.class ),
  @Convert( … )
} )
class …
```

If the data type to be mapped exists several times in the entity, the converter can be attached to the class. In this case, you specify the name of the persistent attribute (`strings`, in our example) and then the assigned converter. In this way, multiple converters can be set locally on this entity.

Note

The Jakarta Persistence specification says that `AttributeConverters` can't be used portably with IDs, version columns, `@Enumerated` columns, temporal data types, and entity references. They can be used well for enumeration elements, and then `@Enumerated` is no longer necessary.

Attribute converters obscure the internal representation, so Native SQL queries must be reformulated.

Task: Given the `enum Gender { FEE, MAA }`, write a `AttributeConverter<Gender, Byte>` that maps `Gender.FEE` to 1 and `Gender.MAA` to 2, so that the converter could be used like this on a persistent attribute:

```
@Converter( GenderConverter.class ) Gender gender;
```

Suggested solution: We can implement the interface like this:

```
@Converter
public class GenderConverter implements AttributeConverter<Gender,Byte> {
  @Override public Byte convertToDatabaseColumn( Gender gender ) {
    return switch ( gender ) { case FEE -> 1; case MAA -> 2; };
  }
  @Override public Gender convertToEntityAttribute( Byte gender ) {
    return switch ( gender ) {
      case 1 -> Gender.FEE;
      case 2 -> Gender.MAA;
      default->throw new IllegalStateException( "Unexpected value: "+gender );
    };
  }
}
```

6.10.9 Key Identification

Every object in Java has an object ID in the JVM, but it's not visible to us. In databases, the primary key (ID) identifies the row. The special persistent attribute of an entity bean with the primary key is annotated with `@Id`:[31]

```
@Id
private Long id;
```

The entity bean will get a key and should never change this key. Of course, there shouldn't be two entity beans with the same key.

Supported Key Types

Jakarta Persistence supports various key types. These include primitive values (e.g., `long`) and the corresponding wrapper classes (e.g., `Long`), which can also be `null`. Strings and data values can be used, as well as `BigInteger` and `BigDecimal`. Since Jakarta Persistence 3.1, which Spring Boot 3 supports via Hibernate ORM 6.1, `UUID` is also supported as a key type. The UUID keys are 128 bits wide and, depending on the database, can result in a performance penalty over single-increment numeric data types.

In principle, you can also build a self-defined key type to map composite keys.

Set Key

Jakarta Persistence needs to support different types of keys. There are *natural keys*, which identify a row in a "natural" way, such as by the email address, nickname, or ISBN. On the other hand, keys can also be *artificial* such as a number or a UUID, for example. In this case, we have the option of assigning a UUID ourselves and then storing the row under this UUID. Another possibility is that the server generates an automatic key, also known as a *surrogate key*.

In the case of the `Profile` table, we have two keys: the email address is a natural key of the row, and there is a surrogate key for the ID. Using a surrogate key has several advantages. Natural keys can potentially change, which can cause problems with foreign key relationships. Additionally, a natural key that occurred only once in the original modeling could suddenly appear several times in a remodeling. Especially with distributed databases, it's not unusual for the program to assign UUIDs instead of the database generating the key. This approach is advantageous because documents can be inserted into the database at various points without conflicts during insertion. Otherwise, there would be a bottleneck in the unique assignment of IDs.

---

[31] *https://jakarta.ee/specifications/persistence/3.1/apidocs/jakarta.persistence/jakarta/persistence/id*

@GeneratedValue

Surrogate keys generated by the database are useful and generally performant. The entity bean marks database generated values with `@GeneratedValue`:[32]

```
@Id
@GeneratedValue
Long id;
```

The annotation is important when saving the rows because the column with the automatically generated value isn't included in the SQL `INSERT`.

@GeneratedValue with GenerationType for a Strategy

A database can use different strategies for automatically generated values. `@GeneratedValue` allows specifying a `strategy` of type `GenerationType` (an `enum`) for how the key is generated. The assignment is `GenerationType.AUTO` by default, which looks like this when fully written:

```
@GeneratedValue( strategy = GenerationType.AUTO )
```

Let's take a closer look at the different types:

- **`SEQUENCE`**
  This is a method of generating an ID sequence that is supported by most databases. The generated value is independent of any specific table, ensuring that two different tables never receive the same key. However, there may be jumps in the IDs during insert operations on a table.
- **`IDENTITY`**
  This is another method of generating keys using an identity column. Unlike `SEQUENCE`, the generated value is specific to a particular table, which means that different tables could potentially receive the same key using `IDENTITY`. Additionally, batch inserts may not be as efficient with `IDENTITY` as they are with `SEQUENCE`.
- **`TABLE`**
  The database doesn't assign the ID, but the software increments the ID. For this, the

---

[32] *https://jakarta.ee/specifications/persistence/3.1/apidocs/jakarta.persistence/jakarta/persistence/generatedvalue*

last stored key is saved in a table. This approach is portable, which is a benefit, but it can also be slow due to the need for multiple read/write accesses.

- `UUID`
  This selects a random UUID. This is interesting for databases in the cloud because this strategy is stateless, so the last value doesn't need to be stored. This is new in Jakarta Persistence 3.1.
- `AUTO`
  This is used with `TABLE`, `SEQUENCE`, or `IDENTITY`, depending on what the database supports. This is an acceptable default value. There are different methods for generating a UUID and the standard does not specify which one must be used.
- `NONE`
  This isn't set automatically.

The different Jakarta Persistence providers offer additional generators, for example, to set a prefix.[33]

Composite Keys with Multiple @Id and @IdClass Annotations

In some cases, there may not be just one primary key for a table. Instead, the primary key may be composed of multiple columns. This is known as a *compound primary key*. Jakarta Persistence supports the use of compound keys, although single column primary keys are generally more efficient. However, in some cases, such as with legacy databases, a compound primary key may be necessary.

There are different ways of mapping.

Variant 1: `@IdClass`

One approach is to apply the `@Id` annotation to the relevant persistent attributes, such as `a` and `b` in the following example:

```
@Entity
class MyEntity {
  @Id String a;
  @Id String b;
}
```

In the next step, the two keys must be combined into an *ID object*. This is important because the Jakarta Persistence API doesn't allow us to pass two or more different columns; so `find(…)` has only one parameter for the key. Therefore, a container is needed that represents the key as an independent object. The class you write for this is called *ID class* and looks something like this:

---

[33] See, for example, *https://stackoverflow.com/questions/18205574/difference-between-generated-value-and-genericgenerator*.

```
public class MyCompositeKey implements Serializable {
  private String a, b;
  public String getA() { return a; }
  public void setA( String a ) { this.a = a; }
  public String getB() { return b; }
  public void setB( String b ) { this.b = b; }
  // equals() and hashCode() must be implemented!
}
```

According to the specification, the ID class must implement the `Serializable` interface, be `public`, and have a public parameterless constructor. For the persistent attributes in the entity bean annotated with `@Id`, there are the same persistent attributes or setters/getters in the ID class—the access type of the entity bean transfers to the ID class. In our example, persistent attributes `a` and `b` belong to the key and therefore there are also instance variables `a` and `b` in the ID class. `equals(…)` and `hashCode()` must also be implemented.

Furthermore, annotation `@IdClass` is added to the entity bean,[34] which refers to the ID class:

```
@Entity
@IdClass( MyCompositeKey.class )
class MyEntity …
```

Variant 2: `@EmbeddedId` **and** `@Embeddable`

Another approach is to use the ID class directly as the type, rather than annotating the columns with `@Id` in the entity bean:

```
@Entity
class MyEntity {
  @EmbeddedId
  private MyCompositeKey id;
  …
}
```

There are then no longer several persistent attributes in the entity bean that are annotated with `@Id`, but the variable for the ID object is annotated with `@EmbeddedId`.[35] The annotation `@IdClass` on the class is no longer necessary. If you go this way, the ID class must still be annotated with `@Embeddable`:[36]

---

[34] *https://jakarta.ee/specifications/persistence/3.1/apidocs/jakarta.persistence/jakarta/persistence/idclass*

[35] *https://jakarta.ee/specifications/persistence/3.1/apidocs/jakarta.persistence/jakarta/persistence/embeddedid*

[36] *https://jakarta.ee/specifications/persistence/3.1/apidocs/jakarta.persistence/jakarta/persistence/embeddable*

```
@Embeddable
public class MyCompositeKey implements Serializable { … }
```

6.10.10 Embedded Types

In entity beans, it's common to have a straightforward mapping between a column and a persistent attribute. However, sometimes a complex element needs to be split across multiple columns. For instance, consider a table with latitude and longitude columns storing a position. In this case, it would be useful to have a compound object, such as a geocoordinate, on the Java side.

Suppose we have a `Unicorn` table with columns for `id`, `email`, `password`, and `profile_fk`. For demonstration purposes, let's assume we want to combine the `email` and `password` columns into a new data type called `LoginCredentials`. Jakarta Persistence offers support for mapping multiple columns to one data type using embedded types. This approach doesn't require entity relationships, but it enables a lean yet OO implementation. The physical structure of the database remains the same, but from the Java perspective, the result looks more OO.

LoginCredentials as @Embeddable

When combining multiple persistent attributes (columns) to create a new data type, the initial step is to create a new container class. In the case of `LoginCredentials`, the container class can be created as follows:

```
@Embeddable
public class LoginCredentials {
  private String email;
  private String password;
  // Setter + Getter
}
```

This data type isn't a standalone entity bean, so the class isn't annotated with `@Entity`. Furthermore, this type isn't independently identifiable by its ID because `@Embeddable`[37] is always associated with a specific entity, in our case, the entity class `Unicorn`.

Embedded with @Embedded

After the declaration of the data type, `@Embeddable` is referenced in the entity bean. For this, the embedded component is annotated with `@Embedded`. The old access methods (i.e., `setEmail(…)`, `getEmail()`, `setPassword(…)`, `getPassword()`) are removed, and new setters/getters for `LoginCredentials` are added:

---

[37] *https://jakarta.ee/specifications/persistence/3.1/apidocs/jakarta.persistence/jakarta/persistence/embeddable*

```
@Entity
@Access( AccessType.FIELD )
public class Unicorn {

  @Id @GeneratedValue( strategy = GenerationType.IDENTITY )
  private Long id;

  private String email;

  @Embedded
  private LoginCredentials credentials;

  // Setter + Getter
}
```

If we model an immutable data type, we don't write a setter.

Shared @Embeddable and @AttributeOveride

An `@Embedded` data type is convenient, especially because it can be used for other tables, that is, other entities. However, in other tables, the columns may have different names. In our case, the email address is stored in the `email` column and the password in the `password` column; in other tables, however, the column may just be called `mail` and the password may be `pass` or `pwd`.

`@Embeddable` can also be used with other names, but the `@AttributeOverride` annotation must indicate a rename.[38] However, the embedded object doesn't know about this, so the annotation is written to the persistent attribute in the entity bean class, for example, like this:

```
@Embedded
@AttributeOverrides( {
  @AttributeOverride( name = "email",                    ❶
                      column = @Column( name="mail" ) ) ❷
} )
private LoginCredentials credentials;
```

❶ For the entity bean, the persistent attribute `email` will be mapped to the column `mail`.

❷ Column `mail` is being mapped to.

This way, `LoginCredentials` can still be used, and we have a renaming of the corresponding columns.

---

[38] *https://jakarta.ee/specifications/persistence/3.1/apidocs/jakarta.persistence/jakarta/persistence/attributeoverride*

@ElementCollection

Mapping data types with `AttributeConverter`, as discussed in Section 6.10.8, can be used to store a list of strings separated by commas. This is a simple implementation of a special 1:n association. Jakarta Persistence offers two “real” options for mapping 1:n relationships:

- **`@ElementCollection`**
  In this case, the children are stored in another table, and there is a foreign key relationship. The parent entity bean manages basic or embedded types in a data structure. The disadvantage of `@ElementCollection`[39] is that these child elements don’t have their own lifecycle, but are always attached to the parent. If an element is deleted or added from this `@ElementCollection`, all elements in the second table are completely deleted, created, and overwritten. This is an expensive operation, which is why it’s generally recommended to avoid `@ElementCollection`.
- **`@OneToMany`**
  This second option is generally more performant, and the entities can be queried and deleted independently. There is no exclusive binding of the children to a parent. We’ll take a closer look at this type of relationship in Section 6.11.

6.10.11 Entity Bean Inherits Properties from a Superclass

In an entity bean, inheritance relationships are possible in principle. However, two types of inheritance must be distinguished:

- **`@Inheritance`**
  Inheritance on the Java side is reflected on the database side.
- **`@MappedSuperclass`**
  The type relationship exists only on the Java side, not on the database side.

There are no true inheritance relationships on the database side, and Jakarta Persistence provides three strategies for mapping Java inheritance to the database. However, because this is an uncommon feature, it’s not relevant for this book.

@MappedSuperclass

With `@MappedSuperclass`,[40] type relations can be expressed on the Java side. This allows us to specify that the ID column is declared in a superclass, such as in the following example, where a superclass declares an ID of type `Long`:

```
@MappedSuperclass
public abstract class AbstractEntity {
```

---

[39] *https://jakarta.ee/specifications/persistence/3.1/apidocs/jakarta.persistence/jakarta/persistence/elementcollection*

[40] *https://jakarta.ee/specifications/persistence/3.1/apidocs/jakarta.persistence/jakarta/persistence/mappedsuperclass*

```
  @Id @GeneratedValue
  private Long id;

  public Long getId() { return id; }
  protected void setId( Long id ) { this.id = id; }
}
```

The abstract superclass is annotated with `@MappedSuperclass`. In this class, there is an ID column as a persistent attribute. The usual annotations `@Id` and `@GeneratedValue` can be found again. With the getter, the ID can be queried from outside, and subclasses should be able to modify the ID. Therefore, the setter is `protected`.

A subclass, such as `Unicorn` or `Profile`, can inherit from this superclass and gets all persistent attributes of the superclass inherited:

```
@Entity
// @Access( AccessType.FIELD )
public class Unicorn extends AbstractEntity {
  …
}
```

Basic Types of Spring Data JPA

The Spring Data JPA project declares an abstract class that can store an ID for us (see Figure 6.9).

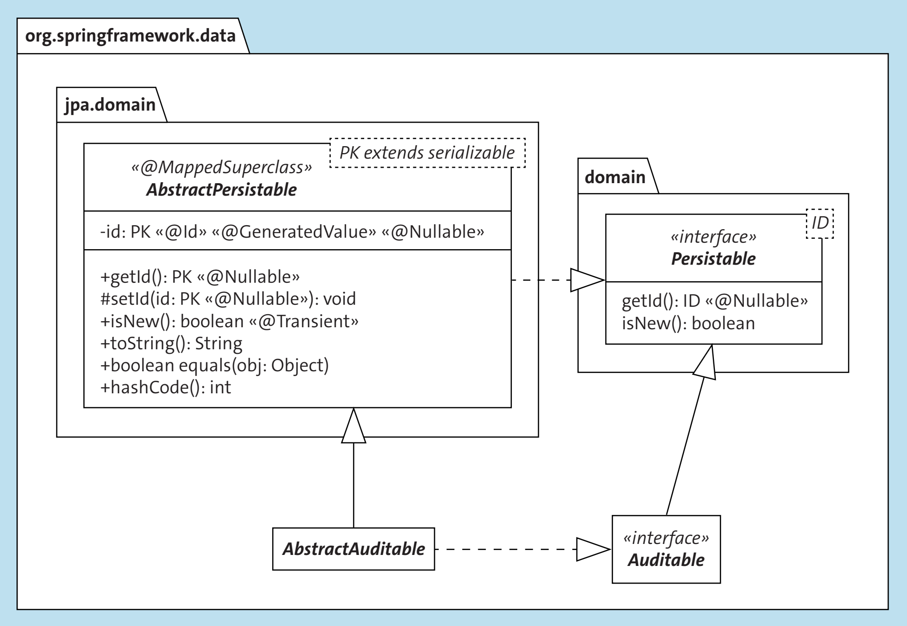

org.springframework.data
jpa.domain
PK extends serializable
«@MappedSuperclass»
AbstractPersistable
-id: PK «@Id» «@GeneratedValue» «@Nullable»
+getId(): PK «@Nullable»
#setId(id: PK «@Nullable»): void
+isNew(): boolean «@Transient»
+toString(): String
+boolean equals(obj: Object)
+hashCode(): int
domain
ID
«interface»
Persistable
getId(): ID «@Nullable»
isNew(): boolean
AbstractAuditable
«interface»
Auditable

Figure 6.9 Abstract Base Types for Custom Entity Beans

Abstract superclass `AbstractPersistable`[41] is a more complex variant of our just written superclass `AbstractEntity` because interface `Persistable`[42] is also used as an abstraction.

In class `AbstractPersistable`, there is subtype `AbstractAuditable` for auditing, which can be used to predefine certain columns, such as where the time of the last changes can be stored. Chapter 7, Section 7.14, introduces the topic of auditing in more detail.

6.11 Relationships between Entities

In the upcoming sections, we'll discuss the relationships between entities. Currently, we only have one entity, the `Profile`. However, if we examine our database tables, we'll see that there are other tables such as `Unicorn` or even `Photo`. The challenge is to determine the most convenient way to establish relationships between these tables.

6.11.1 Supported Associations and Relationship Types

Expressing relationships in Jakarta Persistence is generally straightforward. However, the challenge lies in mapping table relationships into the Java world. In the database tables, there are foreign key relationships, which are columns that refer to other rows. In our case, these references are represented as `long`. To achieve real object references on the Java side, these references shouldn't be expressed as `long` data types. Rather, we want object references that allow us to navigate between different objects.

Relationships can be characterized differently and also by the cardinality. There are 1:1 relationships, but also n:1, 1:n, and n:m relationships. Jakarta Persistence supports the following logical relationship types through annotations:

- 1:1 → `@OneToOne`
- n:1 → `@ManyToOne`
- 1:n → `@OneToMany`
- n:m → `@ManyToMany`

Furthermore, relationships can be either bidirectional, where each side knows about the other, or unidirectional, where only one side knows about the other. Bidirectional relationships are advantageous as they allow for easy navigation from either side to the other.

6.11.2 1:1 Relationship

We have a 1:1 relationship with the `Unicorn` and `Profile` tables. Here, a `Unicorn` entity is related to exactly one `Profile` entity. Table 6.4 shows a small section of the `Unicorn` table:

---

[41] *https://docs.spring.io/spring-data/jpa/docs/current/api/org/springframework/data/jpa/domain/AbstractPersistable.html*

[42] *https://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/domain/Persistable.html*

id	email	password	profile_fk
1	fillmore.fat@wyman.co	{noop}u87szdzwr6j	**1**
2	candy.kane@mills.info	{noop}mk8suwi4kq	**2**

Table 6.4 Extract from the "Unicorn" Table

The `Unicorn` table contains an ID, email address, password, and reference to the `profile` via a foreign key relationship of the `profile_fk` column. The `Unicorn` with ID 1 references the profile with ID 1 (see Table 6.5).

id	birthdate	nickname	manelength	gender	attracted_to_ gender	...
1	1980-08-16	FillmoreFat	3	1	0	...
2	1980-08-01	CandyKane	20	1	1	...

Table 6.5 Extract from the "Profile" Table

It's a coincidence that the ID of the `Unicorn` is the same as the ID of the `Profile`: there is no connection.

To establish a 1:1 relationship between the `Unicorn` and `Profile` tables, we need an entity bean `Unicorn`.

```
@Entity
@Access( AccessType.FIELD )
public class Unicorn {

  @Id @GeneratedValue( strategy = GenerationType.IDENTITY )
  private Long id;

  private String email;

  private String password;

  @Column( name = "profile_fk" )
  private Long profile;

  public Unicorn() {}

  // Setter/Getter
}
```

Listing 6.13 Unicorn.java

The instance variable `Long profile` currently "points" to the profile. You could load a `Unicorn` with `find(…)`, grab the number with `getProfile()`, and use `find(…)` again to load the `Profile`. But that's not the goal. The goal is to get away from the `long` and really reference a `Profile` object.

Owning Side

Associations in Jakarta Persistence (like the other metadata) are usually described by annotations:

- Logical annotations ("It's a relationship!")
- Physical annotation ("How is the relationship realized?")

The question is, to which side does the metadata go? The annotations go to the page with the foreign key; it's called the *owning side*. In our case, the `Unicorn` table "owns" the foreign key for the reference to the `Profile`. `Unicorn` is the owning side, so the annotations must be set there.

For the `Unicorn`, we rewrite our code.

```
class Unicorn {
  …
  @OneToOne                              ❶
  @JoinColumn( name = "profile_fk" )     ❷
  private Profile profile;               ❸
  …
}
```

Listing 6.14 Unicorn.java Extension

❶ A logical annotation `@OneToOne` expresses the 1:1 relationship.

❷ `@JoinColumn` determines the physical mapping; `@Column` isn't allowed!

❸ We use a Java association instead of `Long`.

Annotation `@OneToOne`[43] doesn't indicate how the relationship is realized on the database side. Therefore, another annotation is necessary in addition to `@OneToOne`. Because the database uses a join column—a join table would also be conceivable—we put `@JoinColumn`[44] instead of `@Column`. The Java type changes from `Long` to `Profile`, and when `EntityManager` loads a `Unicorn`, the Jakarta Persistence provider will automatically co-load and build a `Profile` instance. The automatic co-loading is called *eager loading*, and we'll have to deal with this in a bit more detail later in Section 6.12.

To determine the birthday of `Unicorn` 1, we need to look at its associated `Profile`:

---

[43] *https://jakarta.ee/specifications/persistence/3.1/apidocs/jakarta.persistence/jakarta/persistence/onetoone*

[44] *https://jakarta.ee/specifications/persistence/3.1/apidocs/jakarta.persistence/jakarta/persistence/joincolumn*

```
Unicorn fillmoreFat = em.find( Unicorn.class, 1L );
log.info( "profile: {}", fillmoreFat.getProfile().getBirthdate() );
```

The `EntityManager` method `find(…)` loads the `Unicorn` and the `Profile` at the same time, which is why object navigation is possible directly with the initialized objects.

SQL for the 1:1 Relationship

The 1:1 relationship automatically results in an SQL join. Hibernate maps this as follows:

```
select
    u1_0.id,
    u1_0.email,
    u1_0.password,
    p1_0.id,
    p1_0.attracted_to_gender,
    …
    p1_0.manelength,
    p1_0.nickname
from
    unicorn u1_0
left join
    profile p1_0
        on p1_0.id=u1_0.profile_fk
where
    u1_0.id=?
```

The join between the `Unicorn` table and the `Profile` table is evident, and it's worth noting that the entity bean `Profile`'s information is automatically loaded alongside the `Unicorn`'s information. This means that there's no need for two separate SQL queries, as everything is obtained in a single request.

JPQL with Association

Up to this point, we haven't noticed a significant difference between JPQL and SQL. However, one advantage of JPQL is that it allows us to navigate through associated elements, which can be very useful in avoiding extra joins.

In this example, we're looking for all `Unicorn`s whose associated `Profile`s have set `2` at the `gender`:

```
em.createQuery( "SELECT u FROM Unicorn u WHERE u.profile.gender = 2",
                Unicorn.class )
  .getResultList()
  .forEach( unicorn -> log.info( unicorn.toString() ) );
```

If we were using SQL, we would need to use a join, whereas with JPQL, the Jakarta Persistence provider takes care of generating the join for us. When using Hibernate, the following SQL is generated:

```
select
    p1_0.id,
    …
    p1_0.nickname,
    u1_0.id,
    u1_0.email,
    u1_0.password
from
    profile p1_0
left join
    unicorn u1_0
        on p1_0.id=u1_0.profile_fk
where
    p1_0.id=?
```

Implicit and Explicit Join

Query `SELECT u FROM Unicorn u WHERE u.profile.gender = 2` contains a navigation over associated objects. Such an implicit join can occur at various places in the JPQL expression.

In our first example about `find(…)`, it was different. We didn't work via JPQL, and Hibernate generated a `LEFT OUTER JOIN`; this is an indication that an O/R mapper can use different join types. We can also use explicit joins in JPQL:

- `INNER JOIN` (e.g., `u.profile.gender`)
- `LEFT [OUTER] JOIN`
- `[ LEFT [OUTER] | INNER ] JOIN FETCH`

Explicit Inner Join

The JPQL query `SELECT u FROM Unicorn u WHERE u.profile.gender = 2` can alternatively be written with an explicit inner join:

```
SELECT u
FROM Unicorn u
INNER JOIN u.profile AS profile
WHERE profile.gender = 2
```

6.11.3 Bidirectional Relationships

So far, the relationship between `Unicorn` and `Profile` has been one way, meaning that you can navigate from a `Unicorn` to a `Profile`, but not vice versa. To enable navigation in both directions, a bidirectional relationship is required. Previously, only the owning side (`Unicorn`) was mapped, and `Profile` didn't have any attributes or annotations that indicated a reference to `Unicorn`.

mappedBy

To establish a reference from `Profile` back to `Unicorn`, we need to add the `@OneToOne` annotation to `Profile` as well. However, instead of specifying any physical mapping information, we use the `mappedBy` annotation attribute to indicate that the mapping information is declared on the other side. To create a bidirectional relationship, we need to add the code that follows to `Profile`.

```
public class Profile {
  …
  @OneToOne( mappedBy = "profile" )
  private Unicorn unicorn;

  public Unicorn getUnicorn() …
  public void setUnicorn( Unicorn unicorn ) …
  …
}
```

Listing 6.15 Profile.java Extension

The `mappedBy` annotation attribute specifies the name of the property that represents the association on the other side. This effectively tells the Jakarta Persistence provider to look up the persistent attribute named `profile` (specified in `mappedBy`) in the `Unicorn` entity bean, which represents the other end of the 1:1 association. With this information, it's now possible to navigate in both directions.

After implementing the bidirectional relationship, navigation is possible from both `Unicorn` to `Profile` and from `Profile` to `Unicorn`.

```
Unicorn fillmoreFat = em.find( Unicorn.class, 1L );
log.info( "profile: {}",
          fillmoreFat.getProfile().getBirthdate() );

Profile fillmoreFatProfile = em.find( Profile.class, 1L );
log.info( "fillmoreFatProfile: {}",
          fillmoreFatProfile.getUnicorn().getEmail() );
```

6.11.4 1:n Relationship

We established 1:1 relationships in the previous section and saw that relationships can be unidirectional and bidirectional. Another characterization is the *cardinality*. In 1:n relationships, an entity can be related to *n* many entities. These relationships can also be unidirectional and bidirectional.

Java Data Types for 1:n and n:m Associations

Java provides various data structures to store multiple elements. The Collection API includes several data structures that can be used, including the following:

- `java.util.Collection`
- `java.util.Set`
- `java.util.List` (also with order criterion)
- `java.util.Map`

When mapping objects to a database, it's important to consider the key relationships between them and ensure that they are properly represented in the mapping.

Example of a 1:n Mapping in the Relations

Consider two tables in our database that demonstrate a one-to-many relationship: a profile can have many photos associated with it. In the database, the `Photo` table has a foreign key relationship to the `Profile` table. Table 6.6 and Table 6.7 show a small excerpt from each of the tables:

id	birthdate	nickname	manelength	gender	attracted_to_ gender	...
1	1980-08-16	FillmoreFat	3	1	2	...
2	1980-08-01	CandyKane	20	1	1	...

Table 6.6 Rows with IDs 1 and 2 of the "Profile" Table

id	profile_fk	name	is_profile_photo	created
10	1	unicorn010	FALSE	2006-12-06 00:00:00
23	1	unicorn023	FALSE	1998-09-29 00:00:00
11	2	unicorn011	FALSE	2006-04-02 00:00:00

Table 6.7 Rows with IDs 10, 23, and 11 of the "Photo" Table

Each `Profile` and `Photo` in the database has its own unique ID. In the `Photo` table, the `profile_fk` column is a foreign key that references the associated `Profile` table. For instance, photos with IDs `10` and `23` are associated with the `Profile` that has an ID of `1`.

To map this relationship between `Profile` and `Photo` to entity beans, we need to create an entity bean for `Photo`. This listing is a first draft of what the entity bean might look like.

```
@Entity
@Access( AccessType.FIELD )
public class Photo {
  @Id @GeneratedValue( strategy = GenerationType.IDENTITY )
  private Long id;

  @Column( name = "profile_fk" )
  private long profile;

  private String name;

  @Column( name = "is_profile_photo" )
  private boolean isProfilePhoto;

  private LocalDateTime created;

  // Setter/Getter

  @Override public String toString() {
    return "Photo[" + id + "]";
  }
}
```

Listing 6.16 Photo.java

In the draft `Photo` class, the image is represented using a `long`, which is the native data type of the corresponding column in the database. However, as we've seen in our previous work with one-to-one associations, we can refactor the code so that a `Photo` object references a `Profile` object instead.

Photo: The Owning Side and Knows Profiles

As we saw with one-to-one associations, we need to determine which side of the relationship is the owning side for one-to-many associations as well. In our case, the `Photo` table contains the foreign key to the `Profile` table in the `profile_fk` column, making it the owning side.

To build a Java association between `Photo` and `Profile`, we can change the data type of the `profile_fk` field from `long` to `Profile`. We then use the `@ManyToOne` annotation to specify that this is a many-to-one relationship. Because the relationship is realized through a join column in the database, we replace the `@Column` annotation with `@ManyToOne`.[45] With these three changes, the code for our `Photo` entity bean might look like this:

---

[45] *https://jakarta.ee/specifications/persistence/3.1/apidocs/jakarta.persistence/jakarta/persistence/manytoone*

```
class Photo {
  …
  @ManyToOne
  @JoinColumn( name = "profile_fk" )
  private Profile profile;
  …
  public void setProfile( Profile profile ) …
  public Profile getProfile() …
  …
}
```

The setters/getters must also be adjusted with the types; `long` becomes `Profile`.

Now that we've completed the `Photo` entity bean, we can navigate from a `Photo` object to its associated `Profile` object. However, the current unidirectional relationship isn't enough; we also need to be able to navigate from a `Profile` object to its associated photos.

Profile: Knows a List of Photos

We can use the `mappedBy` attribute for 1:n associations as well, just as we did for bidirectional 1:1 associations relationships. In this case, the `Photo` entity has a `@ManyToOne` annotation that refers to the associated `Profile`, and the Profile entity has a `@OneToMany` annotation with `mappedBy` set to `profile` to indicate the relationship with the associated `Photo` objects.

```
class Profile {

  @OneToMany( mappedBy = "profile", fetch = FetchType.EAGER )
  private List<Photo> photos = new ArrayList<>();
  …
  public List<Photo> getPhotos() {
    return photos;
  }
  public Profile add( Photo photo ) {
    photos.add( photo );
    return this;
  }
  …
}
```

Listing 6.17 Profile.java Extension

In addition to the `mappedBy` annotation attribute, the `fetch` attribute specifies that when a `Profile` object is loaded, the associated `Photo` objects should also be fetched. (More about this mechanism follows in Section 6.12.) The photos are stored in a list, which can be sorted, and other `Collection` types such as `Set` or `Map` can also be used.

If we create a new `Profile` object in Java and give it photos directly, the list is initialized. Setters/getters for the photos aren't strictly necessary, and the getter returning the list can lead to modification of the list from outside, which is generally considered an antipattern. Photos can be added to the list from outside using the `add(…)` method. In the case of bidirectional relationships, it may be useful to provide a `this` reference to the other side in the `add(…)` method, but this isn't done in this example.

Load All Photos of a Profile

Now it's easy to obtain photos from a `Profile`:

```
Profile fillmoreFat = em.find( Profile.class, 1L );
List<Photo> photos = fillmoreFat.getPhotos();
log.info( photos.toString() );
```

After loading the profile, a call to `getPhotos()` is sufficient.

The generated SQL shows the following:

```
select
    p1_0.id,
    …
    p1_0.nickname,
    p2_0.profile_fk,
    …
    p2_0.name,
    u1_0.id,
    u1_0.email,
    u1_0.password
from
    profile p1_0
left join
    photo p2_0
        on p1_0.id=p2_0.profile_fk
left join
    unicorn u1_0
        on p1_0.id=u1_0.profile_fk
where
    p1_0.id=?
```

When `EAGER` loading is used, the photos associated with a profile are loaded at the same time as the profile itself. This is achieved with a single SQL statement, which includes both the profile and photo information. Because there can be multiple photos for each profile, the profile information is repeated multiple times for different photos. This redundancy is necessary to fetch all the data in one go using the Jakarta Persistence provider.

Task: List Photos

With this information, we have everything together for a small task. A new command called `photos` should be written, which lists all photos for a given profile ID. The command can be added to the `EntityManagerCommands` class.

Proposed solution:

```
@ShellMethod( "Display all photos of a given profile by ID" )
public void photos( long id ) {
  Optional.ofNullable( em.find( Profile.class, id ) )
          .ifPresent( profile -> {
    for ( Photo photo : profile.getPhotos() )
      System.out.println( photo.getName() );
  } );
}
```

Listing 6.18 EntityManagerCommands.java Extension

Sorting with @OrderBy

This data in the list isn't sorted, but an additional annotation `@OrderBy`[46] can sort the results:

```
@OneToMany( mappedBy = "profile" … )
@OrderBy( "created" )
private List<Photo> photos;
```

The annotation attribute determines the persistent attributes by which the data will be sorted. The Jakarta Persistence provider will generate a SQL statement with a known `ORDER BY`, which means that the sorting is done by the database, not the Java side.

Multiple criteria are possible, like this: `@OrderBy( "isProfilePhoto, created DESC" )`. A trailing `ASC` or `DESC` determines the order. If `@OrderBy` has no set attribute, it's sorted by the primary key.

Note

There are proprietary extensions, for example, from Hibernate, that can sort data structures via a `Comparator`, and that would be software-side. For Hibernate, these are the annotations `@SortComparator` and `@SortNatural`.[47]

---

[46] *https://jakarta.ee/specifications/persistence/3.1/apidocs/jakarta.persistence/jakarta/persistence/orderby*

[47] For more Hibernate annotations, see *https://docs.jboss.org/hibernate/stable/core/javadocs/org/hibernate/annotations/package-summary.html*.

JPQL with 1:n Associations

We've observed that annotations are used to declare associations between entities and that when loading entities via `EntityManager`, referenced entities are also loaded. In addition to the EntityManager API, JPQL can also be used to navigate associations, providing useful functionality. Here are some examples:

- The associated elements can be returned. There are two possibilities with the same result:

```
SELECT p.photos FROM Profile p
SELECT photo FROM Profile p JOIN p.photos photo
```

- A special keyword combination `IS EMPTY` and `IS NOT EMPTY` can test whether collections are empty or not, respectively:

```
SELECT p FROM Profile p WHERE p.photos IS EMPTY
SELECT p FROM Profile p WHERE p.photos IS NOT EMPTY
```

- In addition, it's possible to ask about the size of the collections:

```
SELECT p FROM Profile p WHERE p.photos.size > 6
SELECT p FROM Profile p WHERE SIZE(p.photos) > 6
```

- By the way, you don't necessarily have to use `SIZE(…)`, as a subquery is also possible:

```
SELECT p FROM Profile p WHERE (SELECT COUNT(photo) FROM p.photos photo) > 6
```

  The code using the JPQL function `SIZE(…)` is definitely more concise.

Additionally, JPQL provides other useful functionalities such as accessing elements by index using `INDEX` (only works with `@OrderBy`), accessing the key in case of an associative store using `KEY`, and checking if an element is part of a collection using `[NOT] MEMBER OF`. The Jakarta Persistence specification lists even more JPQL functions, but we've covered most of the possibilities.

6.11.5 n:m Relationships *

We've explored how to map 1:1 and 1:n relationships, but we haven't covered the n:m relationship type yet. This type of relationship is usually implemented on the database side using a *join table*, also known as a *connection table*. Join tables have columns that contain the primary keys of the related tables, in our case, the referenced entities. In our sample database, we have a join table called `Likes`. An excerpt from the table is shown in Table 6.8.

Join table `Likes` contains two columns, `LIKER_FK` and `LIKEE_FK`. The first row in our example indicates that the profile with ID `1` likes the profile with ID 3. By analyzing the data, we can see that profile `1` likes multiple profiles including `3`, `14`, and `15`. The table is useful for determining who likes whom, but it's a directed relationship. Therefore, `1` → `3` only implies that the profile with ID `1` likes the profile with ID `3`, but not the other way around. The snippet of the table also reveals that ID `3` likes the profile with ID `1`, which is great news for Fillmore.

LIKER_FK	LIKEE_FK
1	3
1	14
1	15
2	4
2	7
2	8
3	1

Table 6.8 Sample from the "Likes" Table

Mapping an n:m Relationship in Jakarta Persistence

There are two approaches to mapping n:m relationships in Jakarta Persistence. The first approach is to use the `@ManyToMany`[48] annotation in the entity bean class for `Profile`, similar to how `@OneToOne` or `@ManyToOne` annotations are used. Alternatively, an "artificial" entity bean can be created where the two foreign keys, `liker` and `likee`, become a composite primary key for the entity bean. This "unnatural" entity bean would then have two 1:n associations based on the foreign keys `liker` and `likee`. This approach may be necessary when the join table contains additional data, such as the date of the like event. One of the drawbacks of using `@ManyToMany` is that the join table may only consist of two columns.

6.12 FetchType: Lazy and Eager Loading

When managing associations, the Jakarta Persistence provider operates differently than with simple data types such as strings or numbers. When an entity bean references another entity bean, additional objects need to be created, and reference variables or data structures must be initialized. To load the referenced objects, the Jakarta Persistence provider offers two options:

- **Lazy loading**
  In this case, the referenced objects aren't loaded immediately when the parent entity bean is loaded. Instead, they are loaded on demand when the application code actually accesses them. This approach can be beneficial in cases where the

---

[48] *https://jakarta.ee/specifications/persistence/3.1/apidocs/jakarta.persistence/jakarta/persistence/manytomany*

referenced objects aren't needed all the time, and loading them can be expensive or time-consuming.

- **Eager loading**
  In contrast, eager loading involves loading all the referenced objects at the same time as the parent entity bean. This can be useful when the referenced objects are always needed, and it's more efficient to load them all at once.

6.12.1 Enumeration with FetchType

Enumeration element `FetchType.EAGER`[49] is one of two possible constants; Jakarta Persistence declares `FetchType.LAZY` (late, on first access) and `FetchType.EAGER` (always load).

In the example in Section 6.11.4, we've already used the `FetchType`.

```
@OneToMany( mappedBy = "profile", fetch = FetchType.EAGER )
private List<Photo> photos;
```

Listing 6.19 Profile.java Excerpt

There are different default settings for the associations:

- For `@OneToOne` and `@ManyToOne`, it's `EAGER`.
- For `@OneToMany` and `@ManyToMany`, it's `LAZY`.

In our example, if there is a 1:1 relationship between `Unicorn` and `Profile`, and the `EntityManager` loads a `Unicorn` object, the associated `Profile` object is also loaded by default. However, if a `Profile` object has a collection of photos, these photos aren't automatically loaded when the `Profile` is loaded. This is the default behavior, which is overridden by setting fetch `= FetchType.EAGER` for photos.

Using lazy loading is a good practice because it loads objects only when they are actually needed. This can save a significant amount of data, as unnecessary data doesn't need to be loaded and initialized.

6.12.2 Hibernate Type: PersistentBag

Lazy loading is implemented by Jakarta Persistence providers using a special proxy. This proxy appears to be a data structure with the necessary data, but it actually delays loading the data structure until it's needed. This is why only basic types of the Collection API, such as `Collection`, `List`, `Set`, or `Map`, are allowed in 1:n associations, and concrete implementations such as `ArrayList` or `HashSet` aren't permitted. Hibernate, for instance, prepares its own implementation of this data structure called `org.hibernate.collection.internal.PersistentBag`[50] for the `List` data type.

---

[49] *https://jakarta.ee/specifications/persistence/3.1/apidocs/jakarta.persistence/jakarta/persistence/fetchtype*

Even if an `ArrayList` is built in the code, Hibernate will replace the list with the `PersistentBag` to implement lazy loading:

```
@OneToMany( mappedBy = "profile" /*, fetch = FetchType.LAZY */ )
private List<Photo> photos = new ArrayList<>(); // -> PersistentBag
```

If a method such as `get(0)` or `iterator()` is called on the list (the `PersistentBag`), Hibernate looks to see if data has already been loaded, and, if not, the database access follows.

LazyInitializationException

Lazy loading can be a good choice when it's not certain whether the data will be required in the future. However, it can also result in some challenges. For instance, let's consider a scenario where the default setting of `FetchType.LAZY` is used for our `@OneToMany` association:

```
@OneToMany( mappedBy = "profile" )
private List<Photo> photos;
```

A Java program should load the profile with ID 1 and ask for the number of photos:

```
List<Photo> photos = em.find( Profile.class, 1L ).getPhotos();
photos.size();
```

In the first step, an SQL query would load the profile:

```
select
    u1_0.id,
    …
    p1_0.nickname
from
    unicorn u1_0
left join
    profile p1_0
        on p1_0.id=u1_0.profile_fk
where
    u1_0.profile_fk=?
```

We can see from the output that *no* photo information was loaded, as the photos association is marked as `LAZY`. When method `getPhotos()` is called, it returns `List<Photo>`, which is actually an instance of the `PersistentBag` implementation provided by Hibernate. If the `size()` method is called on this list, Hibernate will attempt to execute a new query to retrieve the size of the collection. However, an exception will occur:

---

[50] *https://docs.jboss.org/hibernate/stable/core/javadocs/org/hibernate/collection/spi/ PersistentBag.html*

```
Caused by: org.hibernate.LazyInitializationException: ↩
failed to lazily initialize a collection of role: ↩
com.tutego.boot.jpa.entity.Profile.photos, ↩
could not initialize proxy - no Session
```

The exception contains a reason: `no Session`. Apparently, the connection was disconnected, so the data could not be reloaded, but why?

After `find(…)`, Spring Data JPA will close `EntityManager`, thus ending the session. In addition, a Jakarta Persistence provider will fail reloading if the session with `EntityManager` has already been closed. So, both queries are basically unrelated.

There are different solutions to address the problem of lazy loading:

- One option is to set `FetchType.EAGER` for the associations, but this isn't optimal if it's uncertain whether all the data will be needed. We used this with the 1:n example with the photos.
- Another solution is to start a transaction and keep `EntityManager` open by using the `@Transactional` annotation in the class or method. However, this approach has the risk of causing long transactions that may block other operations on the database side. It can also lead to memory problems when keeping the session open to a GUI. If we keep the session open to a GUI, this is called *Open Session in View*—and this is an anti-pattern. Especially with many users logged in, this quickly leads to memory problems.
- JPQL can also be used to query data, and it offers the `JOIN FETCH` functionality to load the n-parts of a query. This is a good solution for lazy loading. However, a downside of using JPQL is that the query is hard-coded in the string and not variable. This means that two different JPQL strings are needed, one with `JOIN FETCH` and one without, depending on whether lazy loading is desired or not for a particular scenario.
- To address this issue, an entity graph can be used. This is additional information that can be set for the query to decide individually for each query whether it should be `EAGER` or `LAZY`. This is a practical and effective solution. We'll take a look shortly.

6.12.3 1 + N Query Problem: Performance Anti-Pattern

Following our discussion on `LAZY` and `EAGER` fetching strategies, and having observed data reloading, let's now consider a second common issue. Suppose we have a JPQL query that selects all profiles, and subsequently, a loop iterates over each profile to retrieve the number of photos:

```
var profileQuery = em.createQuery( "SELECT p FROM Profile p",
                                   Profile.class );
```

```
for ( var profile : profileQuery.getResultList() )
  log.info( "{}", profile.getPhotos().size() );
```

Question: Assuming that the fetch type `EAGER` is still set, how many queries to the database are there? There is one query to determine all profiles, and then for *n* profiles, one query each for the images for this profile:

```
select p1_0.id, … p1_0.nickname from profile p1_0
select u1_0.id, … p2_0.name from … left join photo p2_0 …
select u1_0.id, … p2_0.name from … left join photo p2_0 …
select u1_0.id, … p2_0.name from … left join photo p2_0 …
select u1_0.id, … p2_0.name from … left join photo p2_0 …
select u1_0.id, … p2_0.name from … left join photo p2_0 …
select u1_0.id, … p2_0.name from … left join photo p2_0 …
…
```

The issue is commonly referred to as the *N + 1 query problem* or *N + 1 select problem*, although it would be more logical to write it as 1 + N because it starts with a single query that expands to N queries through the code. However, in the literature, the order is usually reversed.

We need to solve the problem.

Solution for the N + 1 Query Problem: JOIN FETCH

Join fetch offers a simple solution:

```
var profiles = em.createQuery(
    "SELECT p FROM Profile p JOIN FETCH p.photos", Profile.class );
for ( var profile : profiles.getResultList() )
  log.info( "{}", profile.getPhotos().size() );
```

With join fetch, the photos associated with the profiles are automatically loaded as well.

Avoiding the Lazy Loading and N + 1 Query Problem with an Entity Graph

The lazy loading problem and the N + 1 query problem are distinct issues, but they share certain similarities. While we've already discussed how to address the lazy loading problem through the use of `FetchType.EAGER` or `JOIN FETCH` in JPQL, both solutions are static and can't be customized at runtime.

To tackle this issue, the Jakarta Persistence offers an *entity graph*, which specifies whether entity relationships should be loaded lazily or eagerly. The benefit of using an entity graph is that the JPQL query remains constant, while the graph can be parameterized for different scenarios. This allows for greater flexibility and more efficient querying.

Build an Entity Graph

There are two ways to build an entity graph.

Option 1: Creation via `EntityManager`:

```
EntityGraph<Profile> entityGraph =
    em.createEntityGraph( Profile.class );
entityGraph.addAttributeNodes( "photos" );
```

With `createEntityGraph(…)`, the root object is specified, and methods such as `addAttributeNodes(…)` add the `EAGER` persistent attributes to be loaded (here, `photos`).

Option 2: The entity graph is declared via annotation `@NamedEntityGraph` at the entity bean (similar to a named query):

```
@NamedEntityGraph(
    name = "eager-photos",
    attributeNodes = {
        @NamedAttributeNode( "photos" )
    }
)
public class Profile …
```

The second option works declaratively via annotations. Unlike in the first option, the type is obvious through the entity bean (`Profile`), but a logical name must be assigned, similar to a named JPQL query.

Apply the Entity Graph

In the application, the entity graph must be referenced; again, there are two ways to do this. The first option is to create `EntityGraph` via `EntityManager`, as we've already seen:

```
EntityGraph<Profile> entityGraph = em.createEntityGraph( … );
```

The second option with named entity graphs is to get them via `getEntityGraph(…)`:

```
EntityGraph< ?   > entityGraph = em.getEntityGraph( "eager-photos" );
```

Unfortunately, there is no generic type information.

`EntityGraph` is then set as `Hint` in the query:

```
var photoQuery = em.createQuery( "SELECT p FROM Profile p",
                                 Profile.class );
photoQuery.setHint( "jakarta.persistence.fetchgraph", entityGraph );

for ( var profile : photoQuery.getResultList() )
  log.info( "{}", profile.getPhotos().size() );
```

This way, it can be dynamically determined for an existing JPQL query whether something should be loaded `LAZY` or `EAGER`.

Tip

In most cases, it's recommended to keep the `FetchType` as `LAZY` to avoid loading unnecessary data, and instead use an entity graph to selectively load data as needed. This approach offers more flexibility than setting the `FetchType` to `EAGER`, as it allows the query to remain the same while still being able to parameterize the entity graph at runtime.

6.13 Cascading

When working with entity beans and their related elements, such as `@OneToOne` and `@OneToMany` mappings, there's a special consideration to keep in mind. It's important to note that a state change (e.g., new, managed, detached, or removed) on an entity bean doesn't automatically cause a state change on its associated elements, regardless of whether the association is unidirectional or bidirectional. For instance, if a `Unicorn` entity references a `Profile` entity, and the `Unicorn` is deleted, the `Profile` isn't automatically deleted as well. Similarly, in a 1:n relationship, if a `Profile` entity is persisted, the state change doesn't propagate to its associated `Photos` entities.

Note

A call to `persist(…)`, `remove(…)`, and so on in the parent must follow a `persist(…)`, `remove(…)`, and so on in the child(ren). Additionally, it's vital to establish valid connections between the entities on the Java side, especially in the case of bidirectional relationships. For example, the Jakarta Persistence provider reconstructs bidirectional relationships when restoring, but not when saving. Not everything in Jakarta Persistence API is automatic.

It can be cumbersome to manually pass the state changes of one entity to its associated entities in a Jakarta Persistence application. This is where *cascading* comes in, which allows the persistence provider to automatically propagate state changes from a parent entity to its associated entity beans.

Cascading is particularly useful in scenarios such as when deleting a profile, which is existentially dependent on its associated photos. In such cases, deleting the profile will trigger the deletion of all associated photos. Similarly, when saving a new profile, any new photos associated with the profile will also be persisted automatically.

6.13.1 CascadeType Enumeration Type

Jakarta Persistence supports multiple cascading types, and these are declared in enumeration type `CascadeType`:[51]

- **`CascadeType.PERSIST`**
  This feature in Jakarta Persistence allows associated entities to be persisted along with the parent entity. Essentially, when a parent entity is persisted and transitions to the managed state, any associated child entities will also undergo the same state change to managed and be written to the database. In simpler terms, `CascadeType.PERSIST` enables the automatic persistence of associated child entities when the parent entity is persisted.
- **`CascadeType.MERGE`**
  This passes `merge(…)`.
- **`CascadeType.REMOVE`**
  This passes `remove(…)`, and is only portable for `@OneToOne` and `@OneToMany`.
- **`CascadeType.REFRESH`**
  This passes `refresh(…)`.
- **`CascadeType.DETACH`**
  This passes `detach(…)`.
- **`CascadeType.ALL`**
  This is a combination of the preceding types in this list. This can be dangerous due to the included `REMOVE`.

Note

The `CascadeType.REMOVE` assignment is dangerous, even indirectly as part of `CascadeType.ALL` because if things go awry, half the database could be gone.

Set CascadeType

Cascading is determined by attribute `cascade` for annotations `@OneToOne`, `@ManyToOne`, `@OneToMany`, and `@ManyToMany`. Actually, it belongs only to `@OneToOne` and `@OneToMany`.

Annotation attribute `cascade` allows an array of `CascadeType`:

```
public @interface *To* {
  …
  CascadeType[] cascade() default {};
  …
}
```

---

[51] *https://jakarta.ee/specifications/persistence/3.1/apidocs/jakarta.persistence/jakarta/persistence/cascadetype*

This way, `CascadeTypes` can be enumerated specifically. `cascade=ALL` is equivalent to `cascade={PERSIST, MERGE, REMOVE, REFRESH, DETACH}`. If `REMOVE` isn't desired, you could set `cascade={PERSIST, MERGE, REFRESH, DETACH}`.

Cascading can come with additional costs, which can be evident in the SQL statements logged in some scenarios. It's essential to note that not setting cascading by default can be an intentional design choice to avoid any unintentional side effects when manipulating related entities.

Regarding `REMOVE`, a single JPQL statement executing a `DELETE` operation can significantly improve performance when deleting a large amount of data. Therefore, it's recommended to use JPQL statements for bulk deletions instead of relying on cascading `REMOVE`.

Unset CascadeType

Let's study a small example of what happens if no `CascadeType` is set, but a deletion is attempted:

```
Profile p = em.find( Profile.class, 20L );
em.remove( p );
```

This code snippet loads the profile with ID `20` and tries to delete the entity bean.

Hibernate generated the following:

```
delete
from
    profile
where
    id=?
```

`DELETE` actually looks good, but the database won't allow it due to a constraint violation caused by the `Unicorn` row having a foreign key relationship to `Profile`:

```
"CONSTRAINT_19B: PUBLIC.UNICORN FOREIGN KEY(PROFILE_FK) REFERENCES ↩
PUBLIC.PROFILE(ID) (CAST(20 AS BIGINT))"
Referential integrity constraint violation: "CONSTRAINT_19B: ↩
PUBLIC.UNICORN FOREIGN KEY(PROFILE_FK) REFERENCES PUBLIC.PROFILE(ID) ↩
(CAST(20 AS BIGINT))"; SQL statement: ↩
delete from profile where id=? [23503-214]
```

(For other RDBMSs, the message may look different in detail.) This means Hibernate doesn't cascade `DELETE` to the photos; it just tries to delete the profile.

There are two possible solutions for deleting a profile:

- Open a transaction and manually delete the photos, the unicorn, and finally the profile.
- Let the Jakarta Persistence provider delete the referenced objects for us. Only the `CascadeType` must be set correctly.

6.13.2 Set CascadeType

In cascading, an operation is transmitted from a parent to its associated elements and the children within a collection. The children of `Profile` are `Unicorn` and `Photos`.

Cascading is possible for unidirectional and bidirectional relationships. Let's put the following listing on the persistent attribute `unicorn` for profiles.

```
@OneToOne( mappedBy = "profile",
           cascade  = CascadeType.ALL )
private Unicorn unicorn;
```

Listing 6.20 Profile.java Extension

The `CascadeType.ALL` setting cascades all state transitions for `profiles` to the `Unicorn`. When the code is executed, a new error message appears. It's again the database that has a problem with the deletion:

```
CONSTRAINT_489: PUBLIC.PHOTO FOREIGN KEY(PROFILE_FK) REFERENCES ↩
PUBLIC.PROFILE(ID) (CAST(19 AS BIGINT))"; SQL statement: ↩
delete from profile where id=? [23503-214]]
```

The rows of the photo table reference profiles. So, in the `Profile` class, the parent, we must also set the `CascadeType` for the `photos`.

```
@OneToMany( mappedBy = "profile",
            fetch    = FetchType.EAGER,
            cascade  = CascadeType.ALL )
private List<Photo> photos = new ArrayList<>();
```

Listing 6.21 Profile.java Extension

A program restart shows the desired result, namely a series of `DELETE` statements:

```
delete from likes where likee_fk=?
delete from likes where liker_fk=?
delete from photo where id=?
delete from unicorn where id=?
delete from profile where id=?
```

The following can be easily inferred from this:

- The reference to the profile to be deleted from the `Likes` table must be removed from both the right and left columns, `likee_fk` and `liker_fk`.
- The photos that belong to a profile must be deleted.
- The unicorn is also deleted because it's associated with the profile through a 1:1 association.
- Finally, the profile itself is deleted.

`CascadeType` is a convenient shortcut, but it's possible to forgo it, and a program can manually handle state transitions.

6.13.3 cascade=REMOVE versus orphanRemoval=true

The `REMOVE` cascading type in Jakarta Persistence causes child entities to transition to the remove state when the parent entity is removed, resulting in their automatic removal as well. However, there are scenarios where child entities may disappear without the parent entity explicitly placing them in the remove state.

- For instance, if a new relationship is established with a different entity bean in a 1:1 association, the old entity bean isn't removed by cascading and is left hanging in the air.
- Similarly, setting the reference to `NULL` in a 1:1 relationship doesn't trigger the `REMOVE` cascading behavior, as the referenced entity bean is merely forgotten, but not actively deleted.

While automatic garbage collection would solve this issue in Java, it's not applicable to databases. To address this, Jakarta Persistence offers a feature that automatically deletes child entities that are no longer referenced (called *orphans*) in such cases. In addition to the `cascade` attribute, which can be set to `CascadeType.REMOVE`, the `orphanRemoval` attribute is available as another annotation attribute in Jakarta Persistence.

@OneTo[One|Many](orphanRemoval = true)

By setting the `orphanRemoval` attribute to true for the `@OneToOne` and `@OneToMany` annotations, Jakarta Persistence allows the automatic deletion of orphaned child entities that are no longer associated with their parent entity. This attribute is a more powerful form of `CascadeType.REMOVE`, as it doesn't require a state change to the remove state for the entity bean.

Essentially, when `orphanRemoval=true`, the Jakarta Persistence provider removes the orphans automatically when the parent entity bean is active, but the children are no longer referenced. This way, `orphanRemoval` provides a convenient way to remove child entities without explicitly having to transition their state to remove.

Note

The associated entity beans form a private parent-child relationship. The children can't move to other entities because they would have been deleted before.

6.14 Repositories

Data storage is a crucial component of any sufficiently large program. However, it's not advisable for a program to directly access data storage by opening files, working with a `DataSource` and the JDBC API, or injecting `EntityManager`. This approach creates a tight coupling between the business logic and the specific data stores. If there are changes in the data storage mechanism, then it requires modifications to the code in multiple places in the business logic.

Instead, it's important to separate the business logic from the physical data store to prevent such issues. The program shouldn't depend on specific data stores, but rather use an abstraction layer that hides the implementation details of the data storage. By doing so, any changes to the storage mechanism can be easily accommodated without modifying the business logic. Thus, it's imperative to treat the management of data storage as a separate task from the business logic of the program.

6.14.1 Data Access Layer

The contemporary approach to software design involves using a data access layer that abstracts away all interaction with underlying physical storage systems. This means that the business logic client will only ever access data stores through a well-defined API, as shown in Figure 6.10. The aim of this design is to decouple the business logic from the implementation details of the storage systems, thereby minimizing the impact of changes made to these systems. By separating the concerns of accessing data and implementing business logic, it's easier to maintain, test, and evolve software systems over time.

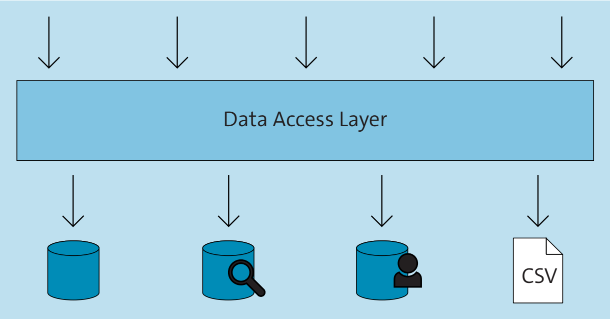


Figure 6.10 Client Accesses Data Stores Only through a Data Access Layer

A data store is a type of system that is used to store and manage data. It can refer to a wide range of technologies, including relational or nonrelational databases, text search engines, directory services such as Lightweight Directory Access Protocol (LDAP), and CSV or XML documents. Essentially, a data store is any system that can hold data in a structured or unstructured format.

One of the key advantages of using a data access layer is that it allows for a clear separation between the business logic and the program code that accesses the data store. These two elements operate at different levels of abstraction and are logically separated from each other. This means that changes to the structure of the data store won't require corresponding changes to the business logic.

By implementing a data access layer, developers can adhere to best practices for writing clean and maintainable code. This includes principles such as *single level of abstraction*, *single responsibility principle*, and *separation of concerns*. Essentially, each module should be responsible for doing one thing well and shouldn't be concerned with other tasks. Additionally, different levels of abstraction shouldn't be mixed. For example, database access should be treated as a completely separate concern from calculations or other functions.[52]

Note: Terms

Although the term "layer" may no longer be the most up-to-date way of conceptualizing an onion architecture, it still serves as a useful model for visualizing the way the different components fit together. In this model, the business logic is situated at the topmost layer, and it interacts with the database layer via the intermediary data access layer.

Repository and Data Access Objects

To leverage the benefits of the data access layer, it's important to delve deeper into its structure. By doing so, we can uncover the *repositories* (as illustrated in Figure 6.11) or the *data access objects* (*DAO*) that are integral components of this layer.

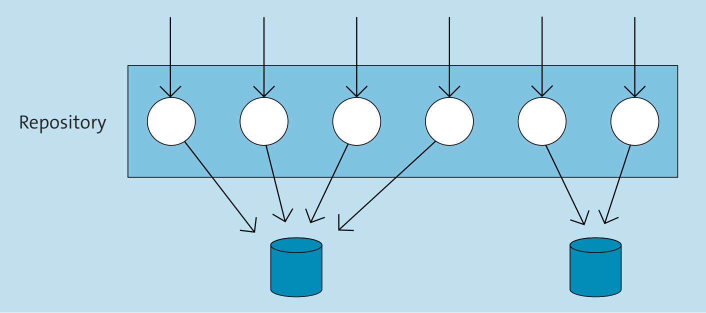


Figure 6.11 Data Access Layer Repositories

A repository in Java is defined as either an interface or a Java class, and its implementation comprises the code responsible for communicating with the relevant data store. The client interacts solely with the Java type and is unaware of the technical implementation details. Specifically, it's only the repository implementation that works with `EntityManager`, for instance.

---

[52] More on the principles of *clean code* can be found online at *https://clean-code-developer.com*, among other places.

Note: Terms

The definitions of the terms *repository* and *DAO* can vary slightly depending on the author. Nonetheless, they share a common objective, which is why we'll use these terms interchangeably and exclusively refer to repositories in the following discussion. Repository originates from *domain-driven design*[53] and is considered a more contemporary term. In older literature, DAO is more commonly used.

6.14.2 Methods in the Repository

Repositories tend to adhere to a consistent basic structure, typically consisting of four key operations: searching for data, creating data, saving data, and potentially deleting data. These operations are commonly referred to as *create, read, update, delete (CRUD) operations*.

Repository API Example from ProfileRepository

To design a repository, it's important to conduct a needs analysis that considers the type of data an application needs to retrieve, store, update, and delete. As a result, the structure of each repository may vary slightly depending on the specific requirements. While some methods may be common to all repositories, others may be unique to a specific application and entity.

For example, a query to retrieve the most popular products from the summer may be relevant for orders but not for addresses. Therefore, such a method would be included in the order repository but not in the address repository.

As an illustration, consider an example of a repository that retrieves unicorn profiles from a database and is also capable of storing them:

```java
public interface ProfileRepository {
  Optional<Profile> findById( long id );
  List<Profile> findAll();
  List<Profile> findByManelengthBetween( int min, int max );
  Profile save( Profile profile );
  long count();
}
```

It's crucial that the methods within the repository are solely intended to serve the business logic's data requirements. The process typically begins with a consideration of which data from the data store will be accessed or modified, followed by the translation of these requirements into Java methods. A Java interface is a convenient means of abstraction.

---

[53] See *Domain-Driven Design: Tackling Complexity in the Heart of Software* by Eric Evans (O'Reilly, 2003).

The presented solution, however, isn't a perfect abstraction because the `Profile` objects are Jakarta Persistence entity beans, which come with known annotations or other metadata. Consequently, it can't be deduced solely from the interface how the implementation has been carried out, although Jakarta Persistence is still evident, indicating the use of a relational database. It's essential to be aware of this type of *leaky abstraction*.

One Conceivable Repository Implementation

For implementation, we would write a class, for example, `JpaProfileRepository`, and implement the interface, that is, program out all the methods:

```
@Repository
public class JpaProfileRepository implements ProfileRepository {

  private final EntityManager em;

  public JpaProfileRepository( EntityManager em ) {
    this.em = em;
  }

  // @Override
  // …
}
```

Via `@Repository`, a Spring-managed bean is created, and via constructor injection, the object gets access to `EntityManager` because the following methods are implemented with it.

Consider the `save(…)` method as an illustration, which is responsible for saving a `Profile`:

```
@Transactional
@Override public Profile save( Profile profile ) {
  if ( profile.getId() == null ) {
    em.persist( profile );
    return profile;
  }
  else {
    return em.merge( profile );
  }
}
```

The `save(…)` method is supposed to be able to save new `Profile` objects and also update existing ones. However, `EntityManager` separates this into two method calls. The question is, has the ID already been set? At the beginning, the `Profile` instance doesn't have an ID yet, so we can call `persist(…)`. The method will store the entity and write the ID to the `Profile` object, which we can return. If the ID has already been set, `merge(…)` will be called, and the object returned by `merge(…)` is what our `save(…)` method will return. The difference is important because `merge(…)` returns a new object, while `persist(…)` updates the passed entity bean. Because `persist(…)` and `merge(…)` are modifying operations in both cases, `@Transaction` must be set either to the method or to the class.

Once the interface is implemented, you can have the repository implementation injected to the desired location via dependency injection:

```
@Autowired
ProfileRepository profiles;
```

At runtime, the client is unaware that it's being provided with a `JpaProfileRepository`. The interface is "neutral," making it agnostic to technology. This aligns with the implementation of the onion architecture, whereby the implementation is situated outside in the infrastructure, while the interface, that is, the repository definition, is located in the core, specifically the domain layer.

6.14.3 SimpleJpaRepository

The repository we've implemented for our profiles is of course different from repositories of other data types, for example, for photos or for messages. Nevertheless, repositories often have things in common, and methods such as `findAll()`, `deleteById(…)`, and `deleteAll()` recur. Because such methods are so general, they can be implemented generically using the `EntityManager` and the Criteria API. We can write helper classes for this, but in the Spring Data JPA, class `SimpleJpaRepository`[54] already does all this for us.

The data type is relatively simple, and we'll look at a small part from the implementation:[55]

```
package org.springframework.data.jpa.repository.support;
import …
@Repository
@Transactional(readOnly = true)
public class SimpleJpaRepository<T, ID>
   implements JpaRepositoryImplementation<T, ID> {
```

---

[54] *https://docs.spring.io/spring-data/data-jpa/docs/current/api/org/springframework/data/jpa/repository/support/SimpleJpaRepository.html*

[55] *https://github.com/spring-projects/spring-data-jpa/blob/main/spring-data-jpa/src/main/java/org/springframework/data/jpa/repository/support/SimpleJpaRepository.java*

```
  public SimpleJpaRepository( Class<T> domainClass, EntityManager em ) {
    this( JpaEntityInformationSupport.getEntityInformation( domainClass,
            em ), em );
  }

  @Override
  @Transactional
  public void deleteAll( Iterable<? extends T> entities ) {
    Assert.notNull( entities, "Entities must not be null" );
    for ( T entity : entities ) {
      delete( entity );
    }
  }

  @Override
  public long count() {
    TypedQuery<Long> query = em.createQuery( getCountQueryString(),
                                             Long.class );
    applyQueryHintsForCount( query );
    return query.getSingleResult();
  }

  @Transactional
  @Override
  public <S extends T> S save( S entity ) {
    Assert.notNull( entity, "Entity must not be null" );
    if ( entityInformation.isNew( entity ) ) {
      em.persist( entity );
      return entity;
    }
    else {
      return em.merge( entity );
    }
  }
  …
}
```

The `SimpleJpaRepository` requires `EntityManager` and type information for the entity bean. Each repository takes care of exactly one type. The type variables `T` and `ID` in `SimpleJpaRepository<T, ID>` stand for the type of the entity and the type of the persistent attribute for the ID column.

The class has quite a few possibilities. I've picked out three methods that show how all methods can be implemented generically, either by methods from `EntityManager` (e.g., `deleteAll()`), `save(…)`, or with the Criteria API as with `count()`.

All methods are transactional by default because the class says `@Transactional(readOnly = true)`. The operations that modify entities such as `save(…)` are `@Transactional`, whereas `readOnly` is `false` by default.

Use SimpleJpaRepository

If we want to use `SimpleJpaRepository`, we only need to pass `EntityManager` and an entity type token in the constructor:

```
var profiles = new SimpleJpaRepository<Profile, Long>(Profile.class, em );
log.info( "{}", profiles.count() );
log.info( "{}", profiles.findAll() );
log.info( "{}", profiles.findAllById( Arrays.asList( 1L, 3L, 5L ) ) );
```

A range of methods already exists. Following is a sample of these options:

- `long count()`
- `void delete(T entity)`
- `void deleteAll()`
- `void deleteAll(Iterable<? extends T> entities)`
- `void deleteAllById(Iterable<? extends ID> ids)`
- `void deleteAllInBatch()`
- `void deleteById(ID id)`
- `boolean exists(Specification<T> spec)`
- `boolean existsById(ID id)`
- `List<T> findAll()`
- `List<T> findAll(Sort sort)`
- `List<T> findAllById(Iterable<ID> ids)`
- `<S extends T> S save(S entity)`
- `<S extends T> List<S> saveAll(Iterable<S> entities)`
- `<S extends T> List<S> saveAllAndFlush(Iterable<S> entities)`
- `<S extends T> S saveAndFlush(S entity)`

Type Hierarchy of SimpleJpaRepository

We can explore the type hierarchy of the `SimpleJpaRepository` class, which is hidden deep in the `org.springframework.data.jpa.repository.support` package (see Figure 6.12).

The class implements numerous interfaces. One important interface is `[List]CrudRepository`, which declares almost all the CRUD methods presented. The `[List]CrudRepository` and also `[List]PagingAndSortingRepository` types come from the *Spring Data Commons package*. The Spring Data JPA implements the `*Repository` interface methods via Jakarta

Persistence, but there are other `*Repository` implementations for other database management systems. We're just scratching the surface.

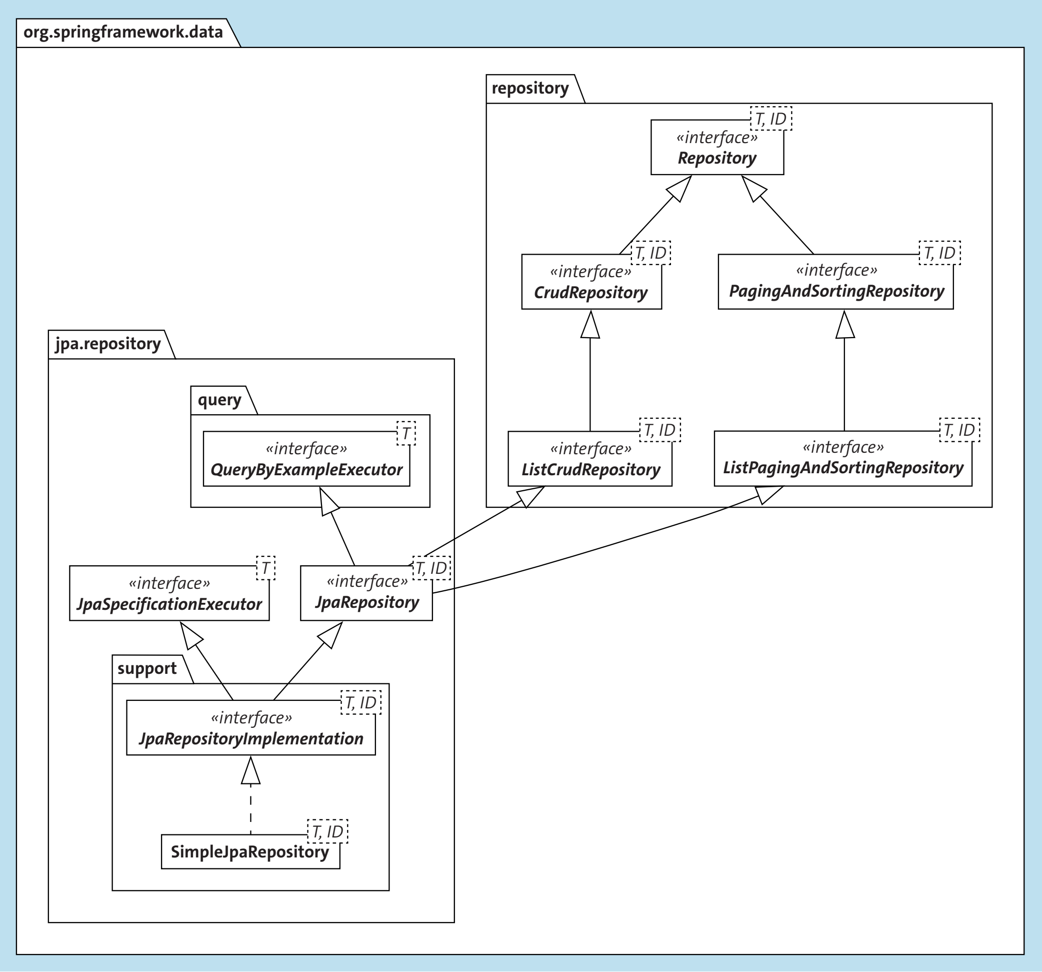

org.springframework.data
repository
«interface» Repository
T, ID
«interface» CrudRepository
T, ID
«interface» PagingAndSortingRepository
T, ID
«interface» ListCrudRepository
T, ID
«interface» ListPagingAndSortingRepository
T, ID
jpa.repository
query
«interface» QueryByExampleExecutor
T
«interface» JpaSpecificationExecutor
T
«interface» JpaRepository
T, ID
support
«interface» JpaRepositoryImplementation
T, ID
SimpleJpaRepository
T, ID

Figure 6.12 Type Relationships of "SimpleJpaRepository"

Do you always have to create `SimpleJpaRepository` instances yourself, or can you request one of the `*Repository` implementations via wiring and then get the implementation as `SimpleJpaRepository`? We'll address these questions in the next chapter.

6.15 Summary

Jakarta Persistence is a comprehensive standard, and the specification at *https://jakarta.ee/specifications/persistence/3.1/jakarta-persistence-spec-3.1.html* is worth reading. However, it may sometimes be necessary to consider the proprietary features of a specific implementation, such as Hibernate ORM. Therefore, a look at the documentation at *https://docs.jboss.org/hibernate/orm/6.1/userguide/html_single/Hibernate_User_Guide.html* is very helpful and a good supplement to the specification.

The primary API we used in this chapter was `EntityManager`. Unfortunately, it's not very comfortable, and a lot has to be done manually. Therefore, we'll now look at an extension that replaces `EntityManager` with something better.

Chapter 7

Spring Data JPA

In this chapter, we want to tackle a new topic, namely *Spring Data*. With Spring Data, repositories for different database systems can be realized quickly and easily.

7.1 What Tasks Does Spring Data Perform?

Each database management system has its own unique application programming interface (API). For example, relational databases are accessed via either Java Database Connectivity (JDBC) or Java Persistence API, while MongoDB and Elasticsearch each have their own distinct API. These APIs often function at a very low level and can differ significantly from one another.

The Spring Data project offers a convenient solution for implementing data access layers. One notable feature is that Spring Data defines its own API, which abstracts away from the details of the underlying database technology. This abstraction allows developers to focus on the business logic, rather than the intricacies of the database management system's API. Additionally, Spring Data provides a wide range of tools and utilities that simplify the development process and reduce the amount of boilerplate code that needs to be written (see Figure 7.1).

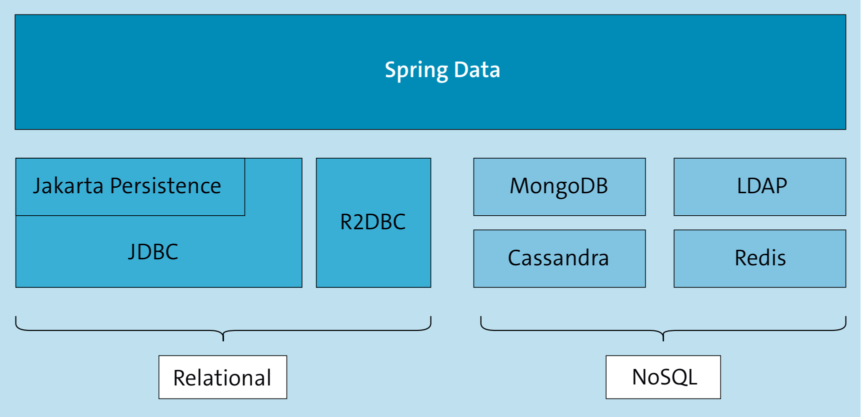


Figure 7.1 Spring Data: One API for All Systems

The development of a common API by Spring Data has significantly simplified the implementation of data access layers. It eliminates the need for developers to learn the

API for each specific database management system, as Spring Data's API unifies both relational and nonrelational databases. This is a remarkable accomplishment that distinguishes Spring Data from the Jakarta Enterprise Edition (Jakarta EE) environment, which only offers the Jakarta Persistence API.

Some notable features of Spring Data include the following:

- Repository interfaces are provided that are similar in syntax for both relational and nonrelational databases.
- Queries can be constructed directly from the method names, which eliminates the need for complex query languages.
- Auditing and enabling tracking of data changes and their timestamps are supported.
- The Spring Data subproject allows for easy publication of REST repositories as REST endpoints.

7.1.1 For Which Systems Is Spring Data Available?

Spring Data can be thought of as a family with different family members. There is the "family API," so to speak, and the individual members of the family implement this API for their central database management system.

Spring Data's website[1] lists the database management systems for which VMware provides implementations. Table 7.1 summarizes this information.

Official Spring Data Modules	Community Modules
Java Persistence API	Neo4j
JDBC	Elasticsearch
MongoDB	Couchbase
KeyValue	Hazelcast
Redis	Azure Cosmos DB
Lightweight Directory Access Protocol (LDAP)	Google Cloud Spanner
Cassandra	. . .
Geode	
GemFire	
Reactive Relational Database Connectivity (R2DBC)	

Table 7.1 Official Modules and Community Modules from the Spring Data Project

---

[1] *https://projects.spring.io/spring-data/*

Quite prominently available as an official module is *Spring Data JPA*, which we use for the examples. However, there is also support from VMware for other important database management systems (e.g., MongoDB and LDAP) or for key-value stores such as Redis.

Some Spring Data implementations aren't developed and managed by VMware, but instead by the vendors of database management systems such as Neo4j, Elasticsearch, and Couchbase. VMware doesn't intend to take on the responsibility or investment for these systems, as the database vendors should maintain them themselves.

Note

This chapter provides an overview of various Spring Data options using Spring Data JPA as a specific example. Even if you're not working with Jakarta Persistence, it's still recommended to read this chapter to gain an understanding of the different data types.

7.2 Spring Data Commons: CrudRepository

Spring Data is responsible for managing entities, and every entity has a unique identifier (ID). This applies to all database management systems, regardless of their type. In a relational database, the ID is typically the primary key of a row. In a document-oriented database such as MongoDB, each entity also has an ID that must be unique.

Spring Data uses a common interface called `CrudRepository` to manage *entities* by providing methods to find, create, update, and delete them. We've already briefly discussed this interface in Chapter 6, Section 6.14.3. The `CrudRepository` is part of the *Spring Data Commons* project and is implemented by concrete database management system–specific repositories. Thus, we don't have to implement the interface ourselves, but instead, Spring Data provides us with an implementation at runtime.

7.2.1 CrudRepository Type

The `CrudRepository` interface in Spring Data is a generic type with two type variables `T` and `ID`:

```
package org.springframework.data.repository;
import java.io.Serializable;

@NoRepositoryBean
public interface CrudRepository<T, ID> extends Repository<T, ID> {
  …
}
```

The first type variable, `T`, is used to indicate the type of entity that the repository is managing. This means that the interface can only manage one entity type at a time, such as a `Unicorn`, a `Profile`, an `Order`, or a `Customer`. The second type variable, `ID`, represents the data type of the entity's unique identifier. This is typically a `long`, a `String`, or a `UUID`.

The `CrudRepository` interface extends the `Repository` interface, which has no methods. The significance of this will be explained in Section 7.3.3.

CrudRepository Methods

The `CrudRepository` itself declares a whole set of methods for CRUD operations, that is, methods for *creating*, *reading*, *updating* and *deleting* entity beans. The following code snippet groups the methods around the following four operations: create and update (to create, save/update), read (operations around reading), and delete (operations that delete):

```
@NoRepositoryBean
public interface CrudRepository<T, ID> extends Repository<T, ID> {
  // C(reate) / U(pdate)
  <S extends T> S save(S entity);
  <S extends T> Iterable<S> saveAll(Iterable<S> entities);

  // R(ead)
  long count();
  boolean existsById(ID id);
  Optional<T> findById(ID id);
  Iterable<T> findAll();
  Iterable<T> findAllById(Iterable<ID> ids);

  // D(elete)
  void delete(T entity);
  void deleteById(ID id);
  void deleteAll(Iterable<? extends T> entities);
  void deleteAll();
  void deleteAllById(Iterable<? extends ID> ids);
}
```

The `save(…)` method saves an entity, and the return is the saved entity. `saveAll(…)` takes an `Iterable`, that is, a collection of entities, such as an `ArrayList`; the result is again an `Iterable` with the saved entities.

The peculiarity of Spring Data is that in `CrudRepository`, there is only one `save(…)` method, and it doesn't matter whether the entity is saved as new or updated. With the *Jakarta Persistence API*, the client must either call the `persist(…)` or `merge(…)` method, as SQL differentiates between saving as new (SQL `INSERT`) and updating (SQL `UPDATE`).

Spring Data JPA internally selects the correct `EntityManager` method for SQL-based databases. Since this differentiation is often not made for other data stores, Spring Data defines a save*(...) API for all possible data stores.

Next, let's look at the read methods: `count()` returns the number of entities. `existsById(…)` determines if an entity exists for a certain key. `findById(…)` tries to load an entity. If the entity isn't found, the result is `Optional.empty()`; otherwise `Optional` is present with the loaded entity. The `findAll()` method returns all entities from the database. The entities to be loaded can also be listed as an `Iterable`; the `findAllById(…)` method returns an `Iterable` of exactly these loaded entities.

Finally, there are several `delete*(…)` methods. The variants are self-explanatory.

These are all predefined methods of the `CrudRepository` interface. In the next step, we need to get a `CrudRepository` from Spring.

7.2.2 Java Persistence API–Based Repositories

In the past, we used the EntityManager API to load entity beans while working with the Spring Data JPA project. However, if you opt for using the Spring Data JPA repositories, you won't need to interact with the EntityManager API. Despite this, the implementation of the CRUD methods still relies on `EntityManager` internally. That's why, in our previous examples, we didn't frequently use the `EntityManager` because Spring Boot applications generally use Spring Data.

Our Own ProfileRepository

We want to declare a repository that provides CRUD operations for profiles. To accomplish this, we write our own interface that inherits from the `CrudRepository` base type. (Other base types exist, which offer some advantages, so we'll look at that too.) Here's how it looks:

```
public interface ProfileRepository extends CrudRepository<Profile, Long> { }
```

The type arguments for generics represent the type of the entity (`Profile`) and the type of the key (`Long`). The peculiarity is that we don't implement our own interfaces; we only declare an interface, which can get further abstract methods, but usually doesn't contain default methods.

If we need to declare an interface and not a class, who implements the interface? This is what Spring Data does at runtime! A Spring Data family member builds an object at runtime that extends `ProfileRepository`, for example. All methods can be called on this `ProfileRepository` implementation. The Spring Data JPA implementation internally realizes the methods with `EntityManager`.

Setting component annotations such as `@Repository` in the interface is unnecessary. These annotations are only required if you create your own repository classes. If we use interfaces, we omit the `@Repository` annotation.

Here's an example of a minimal program with `ProfileRepository`:

```
@SpringBootApplication
public class SpringDataJpaApplication {

  private final Logger log = LoggerFactory.getLogger( getClass() );

  public SpringDataJpaApplication( ProfileRepository profiles ) {
    log.info( "Profile with id=1: {}", profiles.findById( 1L ) );
    log.info( "All profiles: {}", profiles.findAll() );
  }

  public static void main( String[] args ) {
    SpringApplication.run( SpringDataJpaApplication.class, args );
  }
}
```

Via constructor injection, we get the runtime implementation of `ProfileRepository`, and all predefined CRUD methods can be called.

Task: RepositoryCommands with New Shell Methods

If you're interested, you can try writing the code on your own and complete the following task:

1. Create a new class called `RepositoryCommands`, which is again annotated with `@ShellComponent`.
2. Inject the `ProfileRepository` into the new Spring bean `RepositoryCommands`.
3. Write a new command `list` that prints all profiles.
4. Try various `CrudRepository` methods in your own new commands, for example, by building a new `Profile`, storing it in the database, and deleting it later. All modifying methods such as `save(…)` are automatically transactional, as we've seen in Chapter 6, Section 6.14.3.

Tip

`EntityManagerCommands` can be commented out.

Proposed solution:

```
@ShellComponent
public class RepositoryCommands {

  @Autowired
  private ProfileRepository profiles;
  @ShellMethod( "Display all profiles" )
  public void list() {
    profiles.findAll().forEach( System.out::println );
  }

  @ShellMethod( "random" )
  public Optional<Profile> random() {
    Profile generate = new Profile(
        "JuicyJenelyn",
        LocalDate.now().minusYears( 22 ),
        12,
        Profile.FEE,
        null,
        "",
        LocalDateTime.now().minusDays( 12 ) );
    profiles.save( generate );
    return profiles.findById( generate.getId() );
  }

  @ShellMethod( "update" )
  public Optional<Profile> update() {
    Optional<Profile> maybeProfile = profiles.findById( 1L );
    maybeProfile.ifPresent( profile -> {
      System.out.println( profile );
      profile.setNickname( "King" + profile.getNickname() + "theGreat" );
      profiles.save( profile );
    } );
    return profiles.findById( 1L );
  }
}
```

Internal Flow: From Client to Database

The process of accessing a database through the Spring Data JPA project involves several layers of abstraction. These layers include `EntityManager`, which implements the appropriate methods for the repository in the background, and the JDBC driver, which ultimately communicates with the database. These layers of abstraction are in place to allow for flexibility in the choice of database and to shield the client from any changes that may occur.

For example, if a different database were to be used, the client would not need to be aware of it as long as the appropriate `EntityManager` methods are implemented. Similarly, if a different Jakarta Persistence provider were to be used, such as Hibernate ORM or EclipseLink JPA, the client would still be shielded from the changes as long as the provider implements the specification.

It's also possible to replace the JDBC driver, although this isn't recommended as JDBC drivers are typically developed in coordination with the relational database management system (RDBMS) manufacturers. The only API that the client interacts with is the `CrudRepository`, and even if the database management system were to be changed, for example, to MongoDB, the `ProfileRepository` would not need to be modified as long as it's currently programmed with only references to the entity bean class and its annotations, and not to any specific database or persistence implementation.

7.3 Subtypes of CrudRepository

The `CrudRepository` and `Repository` interfaces are very general. There are other interfaces worth inheriting from. We'll discuss a few more basic types in this section.

7.3.1 ListCrudRepository

Whereas the base type for `CrudRepository` issues collections only as `Iterable`, the `ListCrudRepository` subtype [2] (see Figure 7.2) uses a specialization in the return type in the form of lists. When a subtype overrides a method and uses a specialization in the return, this is called a *covariant return type* in Java.

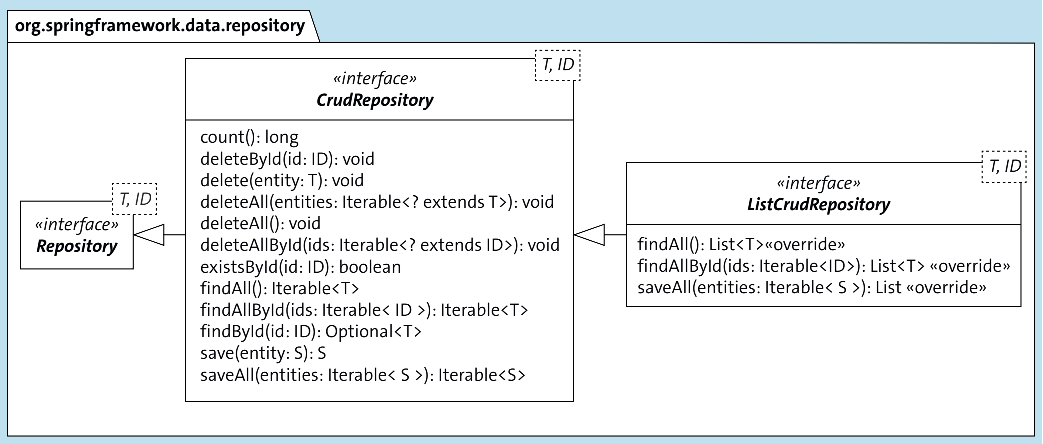

org.springframework.data.repository
T, ID
«interface»
CrudRepository
count(): long
deleteById(id: ID): void
delete(entity: T): void
deleteAll(entities: Iterable<? extends T>): void
deleteAll(): void
deleteAllById(ids: Iterable<? extends ID>): void
existsById(id: ID): boolean
findAll(): Iterable<T>
findAllById(ids: Iterable< ID >): Iterable<T>
findById(id: ID): Optional<T>
save(entity: S): S
saveAll(entities: Iterable< S >): Iterable<S>
T, ID
«interface»
Repository
T, ID
«interface»
ListCrudRepository
findAll(): List<T>«override»
findAllById(ids: Iterable<ID>): List<T> «override»
saveAll(entities: Iterable< S >): List «override»

Figure 7.2 Methods of "ListCrudRepository" Returning "List" Instead of "Iterable"

An `Iterable` solely provides an `Iterator`, which in turn only permits a data source to be processed sequentially. This sequential processing isn't particularly noteworthy or engaging.

---

[2] *https://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/repository/ListCrudRepository.html*

7.3.2 Technology-Specific [List]CrudRepository Subtypes

The `[List]CrudRepository` serves as a universal interface for all Spring Data projects, independent of the specific data store. The methods provided by the repository don't reveal any details about the underlying technology, such as SQL or NoSQL documents, which makes it possible for any database management system to implement `[List]CrudRepository`. However, this can also be considered a disadvantage, as technology-specific features such as batch operations for RDBMSs or weighted text searches are often desired.

Similar to JDBC, Spring Data has an API and various implementations. Spring Data Commons defines common data types such as the `Repository` or `[List]CrudRepository` interfaces, as well as types such as `Sort` or `Pageable`. However, the implementation for a specific data store isn't part of Spring Data Commons but rather the responsibility of Spring Data subprojects, such as Spring Data JPA. These subprojects implement the API and typically declare additional data types, which allows for better optimization of underlying storage systems.

The relationships between the various projects form subtypes of `[List]CrudRepository`, as depicted in Figure 7.3. This design enables more optimal support for different storage systems, such as `JpaRepository` for Java Persistence API or `MongoRepository` for MongoDB.

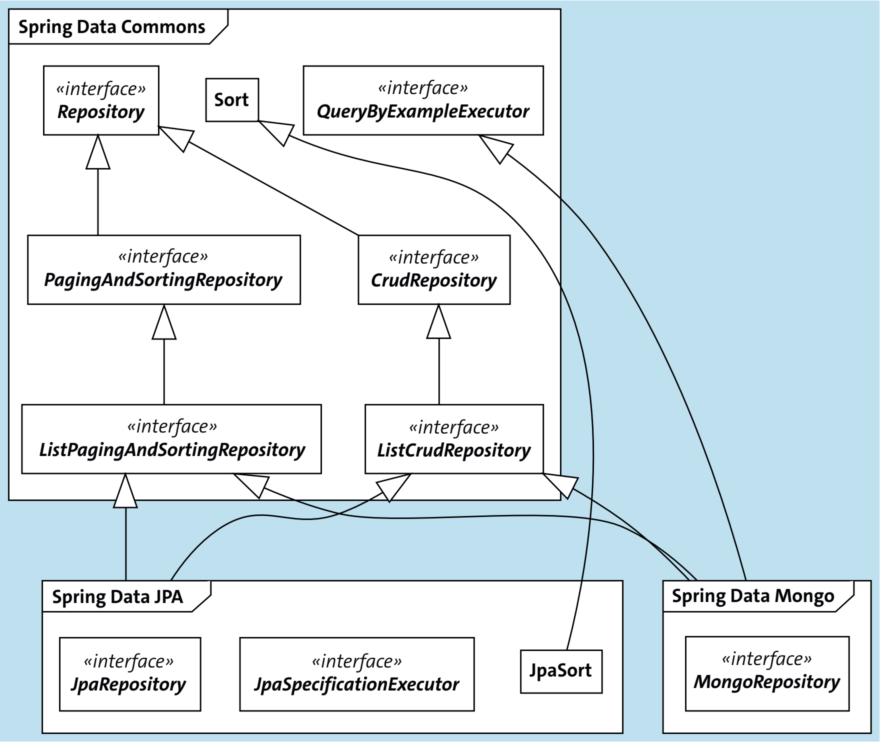

Spring Data Commons
«interface»
Repository
Sort
«interface»
QueryByExampleExecutor
«interface»
PagingAndSortingRepository
«interface»
CrudRepository
«interface»
ListPagingAndSortingRepository
«interface»
ListCrudRepository
Spring Data JPA
«interface»
JpaRepository
«interface»
JpaSpecificationExecutor
JpaSort
Spring Data Mongo
«interface»
MongoRepository

Figure 7.3 Type Relationships of the Spring Data Repositories

Use JpaRepository

In our case, we can “improve” the `ProfileRepository` by using the `JpaRepository` subtype instead of `CrudRepository`.[3] We change

```
public interface ProfileRepository extends CrudRepository<Profile, Long> { }
```

to

```
public interface ProfileRepository extends JpaRepository<Profile, Long> { }
```

Listing 7.1 ProfileRepository.java Extension

JpaRepository Methods and Basic Types

Using `JpaRepository` instead of [`List`]`CrudRepository` has several advantages. The Unified Modeling Language (UML) diagram in Figure 7.4 makes this clear.

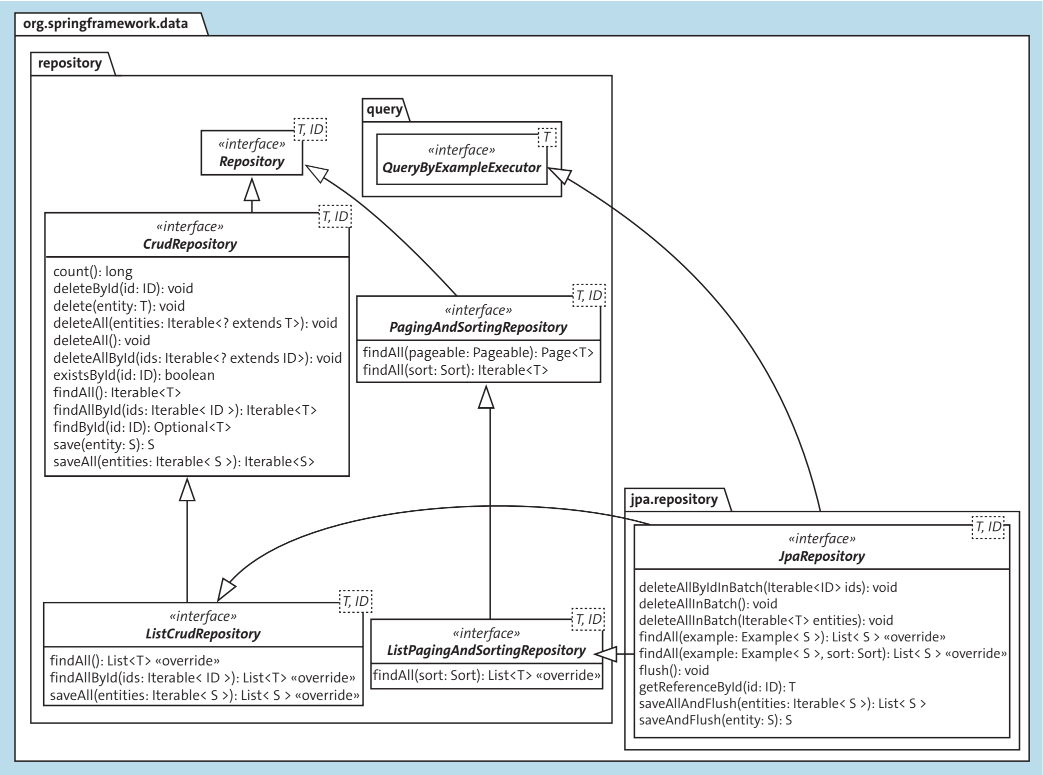

org.springframework.data
repository
query
T, ID
«interface»
Repository
T
«interface»
QueryByExampleExecutor
T, ID
«interface»
CrudRepository
count(): long
deleteById(id: ID): void
delete(entity: T): void
deleteAll(entities: Iterable<? extends T>): void
deleteAll(): void
deleteAllById(ids: Iterable<? extends ID>): void
existsById(id: ID): boolean
findAll(): Iterable<T>
findAllById(ids: Iterable< ID >): Iterable<T>
findById(id: ID): Optional<T>
save(entity: S): S
saveAll(entities: Iterable< S >): Iterable<S>
T, ID
«interface»
PagingAndSortingRepository
findAll(pageable: Pageable): Page<T>
findAll(sort: Sort): Iterable<T>
jpa.repository
T, ID
«interface»
JpaRepository
deleteAllByIdInBatch(Iterable<ID> ids): void
deleteAllInBatch(): void
deleteAllInBatch(Iterable<T> entities): void
findAll(example: Example< S >): List< S > «override»
findAll(example: Example< S >, sort: Sort): List< S > «override»
flush(): void
getReferenceById(id: ID): T
saveAllAndFlush(entities: Iterable< S >): List< S >
saveAndFlush(entity: S): S
T, ID
«interface»
ListCrudRepository
findAll(): List<T> «override»
findAllById(ids: Iterable< ID >): List<T> «override»
saveAll(entities: Iterable< S >): List< S > «override»
T, ID
«interface»
ListPagingAndSortingRepository
findAll(sort: Sort): List<T> «override»

Figure 7.4 “JpaRepository” Super Types and Methods

The first difference is that new methods are added by additional super types. A `JpaRepository` extends `ListCrudRepository` and `ListPagingAndSortingRepository` (we’ll discuss this type in Section 7.4). But this adds two additional methods to `JpaRepository`.

---

[3] *https://docs.spring.io/spring-data/jpa/docs/current/api/org/springframework/data/jpa/repository/JpaRepository.html*

Furthermore, `JpaRepository` inherits from `QueryByExampleExecutor`, which gives another set of additional methods. The methods are hidden in the UML diagram, and we'll get to the data type in Section 7.5.

Furthermore, the `JpaRepository` subinterface has additional methods that aren't present in the base types, such as the batch processing methods. This is because relational databases can efficiently process large amounts of data in batches, and it's helpful to take advantage of this feature. However, other data repositories may not support batch operations, so these methods can't be included in the base type.

There is also a subtle but useful third difference between the base types and the `JpaRepository` subinterface: it overrides two methods from the `QueryByExampleExecutor` interface with covariant return types to return `List` instead of `Iterable`.

Tip

In practice, the base type isn't that important because methods and return types are always present and can be added, as the following section will show.

7.3.3 Selected Methods via Repository

The downside of extending interfaces such as `JpaRepository` or [`List`]`CrudRepository` is that they bring a whole set of methods into your repositories that may not be welcome. In particular, the `delete*(…)` methods may be undesirable if entities should not be deleted.

To "hide" methods, the most general type, `Repository`,[4] comes into play. All `Repository` interfaces in Spring Data are derived from this interface. The base type doesn't declare any methods (see Figure 7.5).

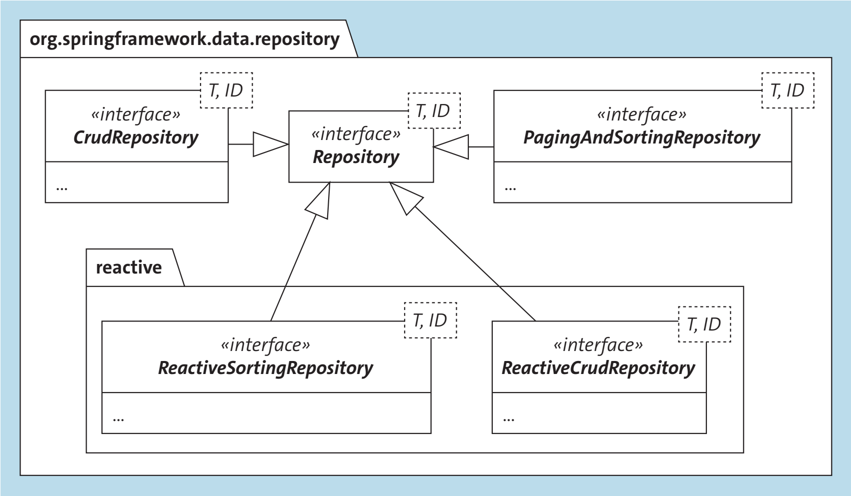

org.springframework.data.repository
T, ID
«interface»
CrudRepository
…
T, ID
«interface»
Repository
T, ID
«interface»
PagingAndSortingRepository
…
reactive
T, ID
«interface»
ReactiveSortingRepository
…
T, ID
«interface»
ReactiveCrudRepository
…

Figure 7.5 UML Diagram of "Repository" and Its Subtypes

---

[4] *https://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/repository/Repository.html*

An interface without CRUD methods makes little sense at first glance, but the trick is this: we extend `Repository` and copy and paste, so to speak, the methods into the interface, which our repository should have. This could look like this:

```
public interface ReadOnlyProfileRepository
    extends Repository<Profile, Long> {
  boolean existsById( Long id );
  List<Profile> findAll();
  List<Profile> findAllById( Iterable<Long> ids );
  Optional<Profile> findById( Long id );
}
```

This form of copy-and-paste programming may not be optimal, but this way, the client only gets the API it needs—this is an application of the Interface Segregation Principle (ISP), which we'll discuss in Section 7.11.2.

Misspellings of methods are detected to some extent. The fact that a repository can have "magic" methods with special names that are then implemented at runtime is a foretaste of Section 7.8.

7.4 Paging and Sorting with [List]PagingAndSortingRepository

This section is about the types `[List]PagingAndSortingRepository`,[5] which were shown previously in Figure 7.5. The data type allows the following:

- "Scrolling" through large result sets, which is also called *pagination*
- Sorting results according to certain properties

The `PagingAndSortingRepository` extends `Repository` (see Figure 7.6).

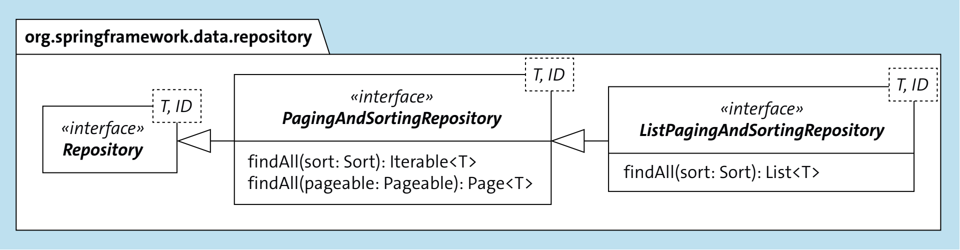

org.springframework.data.repository
T, ID
«interface»
Repository
T, ID
«interface»
PagingAndSortingRepository
findAll(sort: Sort): Iterable<T>
findAll(pageable: Pageable): Page<T>
T, ID
«interface»
ListPagingAndSortingRepository
findAll(sort: Sort): List<T>

Figure 7.6 "[List]PagingAndSortingRepository" Type Relationships

The interface `PagingAndSortingRepository` introduces two new methods: `findAll(Sort)` and `findAll(Pageable)`. Here, `Sort` represents the ordering criteria, and `Pageable` represents the section/block that can be paged forward and backward. Subtype `ListPagingAndSortingRepository`[6] again uses a covariant return type—`List`—instead of `Iterable`.

---

[5] *https://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/repository/PagingAndSortingRepository.html*

7.4.1 JpaRepository: Subtype of ListPagingAndSortingRepository and ListCrudRepository

`JpaRepository` is a subtype of `ListPagingAndSortingRepository` and `ListCrudRepository`, so it combines both interfaces (see Figure 7.7).

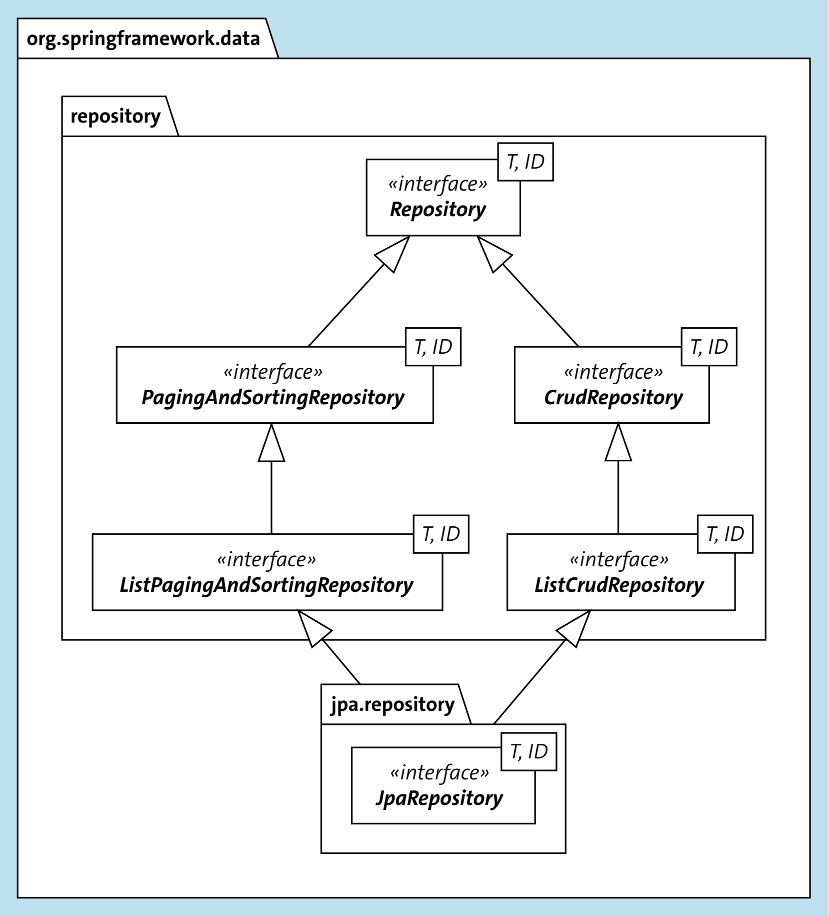

org.springframework.data
repository
T, ID
«interface»
Repository
T, ID
«interface»
PagingAndSortingRepository
T, ID
«interface»
CrudRepository
T, ID
«interface»
ListPagingAndSortingRepository
T, ID
«interface»
ListCrudRepository
jpa.repository
T, ID
«interface»
JpaRepository

Figure 7.7 "JpaRepository": "ListCrudRepository" and "ListPagingAndSortingRepository"

Consequently, our `ProfileRepository` also has the `findAll(…)` methods parameterized with `Sort` and `Pageable`.

---

[6] *https://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/repository/ListPagingAndSortingRepository.html*

7.4.2 Sort Type

To begin, let's consider how to sort query results. `Sort`[7] is a class that has no constructors. Instead, static `by(…)` factory methods build the objects:

- `Sort.by(String... properties)`
- `Sort.by(Order... orders)`
- `Sort.by(List<Order> orders)`
- `Sort.by(Direction direction, String... properties)`

Let's take a look at some examples of building a `Sort` object:

```
Sort sort = Sort.by( "manelength" ).ascending();
```

The simplest way is to pass the name of the persistent attribute to the `of(…)` method. Be careful: with Jakarta Persistence, the name of the persistent attribute is specified, *not* the column name. We won't need the appended `ascending()` because the default is an ascending sort.

Multiple sorting criteria are possible. That is, if the first elements are equal, a second sort criterion can determine the order. One notation looks like this:

```
Sort sort =       Sort.by( "manelength" ).ascending()
            .and( Sort.by( "lastseen"   ).descending() );
```

With `.and()`, another `Sort` object is appended. The result is an ascending sort by the mane lengths, and if two profiles have the same mane lengths, `lastseen` is taken as the second criterion, descending.

The other possibility to initialize `Sort` objects is provided by `Order` objects:[8]

```
Sort sort = Sort.by( Sort.Order.asc(  "manelength" ),
                     Sort.Order.desc( "lastseen" ) );
```

The `Order` class is a nested type of `Sort`. Static methods such as `by(…)`, `asc(…)` (for ascending), and `desc(…)` (for descending) create `Order` objects. The `Order` objects fill the variable argument (vararg) of the `Sort` method `by(Order...)`. The SQL that is generated isn't remarkable at all. Hibernate appends the following to the SQL:

```
order by p1_0_.manelength asc, p1_0_.lastseen desc
```

The big advantage is that we can build the `Sort` object dynamically; think of a spreadsheet and columns that you click to sort. Our Jakarta Persistence Query Language (JPQL) string doesn't change. If the sort were hard-coded in the JPQL string, there would be little flexibility, but the `findAll(…)` method allows the `Sort` object to change the order dynamically.

---

[7] *https://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/domain/Sort.html*

[8] *https://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/domain/Sort.Order.html*

Error with Wrong Property Names

The Jakarta Persistence provider checks the `Sort` object to see if it's a valid persistent attribute. If a `Sort` object is built with an incorrect name, a `PropertyReferenceException` follows. Consider the following scenario:

```
profiles.findAll( Sort.by( "len" ) );
```

Spring will report the following:

```
Caused by: org.springframework.data.mapping.PropertyReferenceException: ↩
No property 'len' found for type 'Profile'!
```

That is correct because it should have been called `manelength`. The example makes clear that arbitrary order criteria aren't possible, which would be possible in principle in SQL.

The fact that the persistent attributes have to be specified as strings when sorting with `Sort` is a minor inconvenience. There are several ways to get away from these strings. One approach is provided by Spring Data with another data type, `TypedSort`, and another is the subject of Section 7.10.1.

7.4.3 Sort.TypedSort<T>

`TypedSort`[9] already expresses via the type name that it's about a typed sort. `TypedSort` is a nested type and is built as follows:

```
Sort.TypedSort<Profile> sortedProfile = Sort.sort( Profile.class );
```

The idea behind `TypeSort` is clever, which is that we need to call getters on the entity bean. A getter, in turn, is attached to the persistent attribute; if a renaming occurs, it stands out in the source code.

```
Sort sort = sortedProfile.by(
    ( Profile profile ) -> profile.getManelength()
);
```

We can refactor the lambda expression into a method reference. Then, we can concatenate two sort criteria using `and(…)` and execute the sorting process:

```
Sort.TypedSort<Profile> sortedProfile = Sort.sort( Profile.class );
Sort sort = sortedProfile.by( Profile::getManelength )
    .and( sortedProfile.by( Profile::getLastseen ).descending() );
List<Profile> sortedProfiles = profiles.findAll( sort );
```

---

[9] *https://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/domain/Sort.TypedSort.html*

Spring Data JPA generates a proxy for the `Profile Class` object, on which we call the getter. The proxy intercepts the method call and thus knows which persistent properties form the sort criterion.

7.4.4 Pageable and PageRequest

We just discussed the first feature of `[List]PagingAndSortingRepository`, namely that the `findAll(…)` method is overloaded with a `Sort` parameter that sorts the results.

The second method from `[List]PagingAndSortingRepository` is `findAll(Pageable)`. Type `Pageable`[10] is an interface, and the implementation is `PageRequest`[11] (see Figure 7.8).

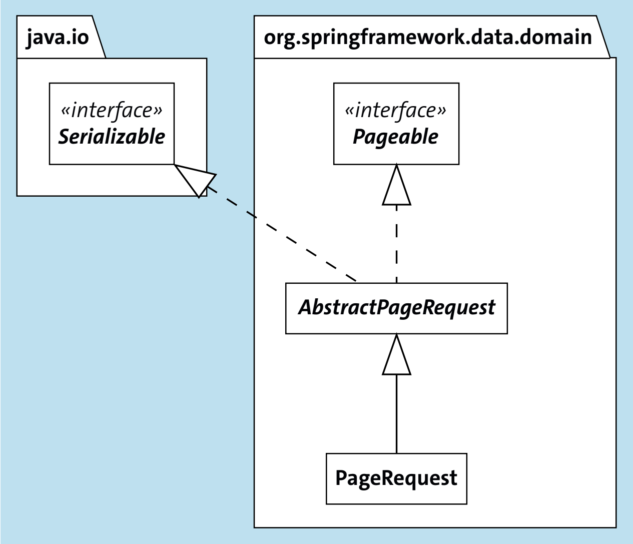


Figure 7.8 Type Relations of "Pageable" and Implementation

There are two ways to build `Pageable` objects: first, via static `of*(…)` methods of the implementing `PageRequest` class, and, second, `Pageable` also has a static factory method that builds a `PageRequest` object in the background:

- `PageRequest.of(int page, int size)`
- `PageRequest.of(int page, int size, Sort sort)`
- `PageRequest.of(int page, int size, Direction direction, String... properties)`
- `PageRequest.ofSize(int pageSize)` → `ofSize(0, pageSize)`
- `Pageable.ofSize(int pageSize)` → `PageRequest.of(0, pageSize)`

---

[10] *https://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/domain/Pageable.html*

[11] *https://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/domain/PageRequest.html*

You need to specify two pieces of information: the page and the number of elements on the page. With pagination, the number of elements per page doesn't normally change, only the page changes. In addition, some `of(…)` methods also let you specify a sort criterion, either via a `Sort` object or via a `Direction` with a vararg for the names of the persistent attributes.

The `PageRequest.ofSize(pageSize)` static method is a shortcut for getting the first page. Consequently, to select the first page with 10 elements, you can write, for example:

```
Pageable pageable = PageRequest.of( 0 /* page */, 10 /* size */ );
```

The return is a `Pageable` object suitable for the `findAll(Pageable)` method.

Page

Method `findAll(Pageable)` accepts a `Pageable` object as an argument, but the method doesn't directly return the results like other `findAll(…)` methods; instead, it returns a `Page` object:[12]

```
Page<Profile> page1 = profiles.findAll( PageRequest.of( 0, 10 ) );
// page1.toString() -> "Page 1 of … containing … instances"
List<Profile> content = page1.getContent();
```

The content of the page is returned by the `getContent()` method. Although this may appear strange, there is a sound rationale behind it.

Run Pages

A `Page` object provides similar functionality to an `Iterator`, allowing you to traverse through the pages. It offers navigation options such as moving to the next or previous page and determining whether it's the first or last page. All of these navigation features are included in the `Page` object or its base type. To illustrate, consider the following example:

```
Page<Profile> page1 = profiles.findAll( PageRequest.ofSize( 10 ) );
Page<Profile> page2 = profiles.findAll( page1.nextPageable() );
```

From a page, `nextPageable()` navigates to the next page. The `nextPageable()` method returns a `Pageable` that can be passed back to `findAll(…)`. With this construction, it's possible to run over all pages with a certain size:

```
Pageable pageRequest = Pageable.ofSize( 10 );
Page<Profile> profilePage;
do {
  profilePage = profiles.findAll( pageRequest );
```

---

[12] *https://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/domain/Page.html*

```
  log.info( "{}", profilePage.getContent() );
  pageRequest = pageRequest.next();
} while ( ! profilePage.isLast() );
```

As mentioned in Section 7.4.5, this approach may not be highly performant, but it's feasible from a technical standpoint.

Page Type Relations

Let's take a look at `Page`'s type relationships (see Figure 7.9).

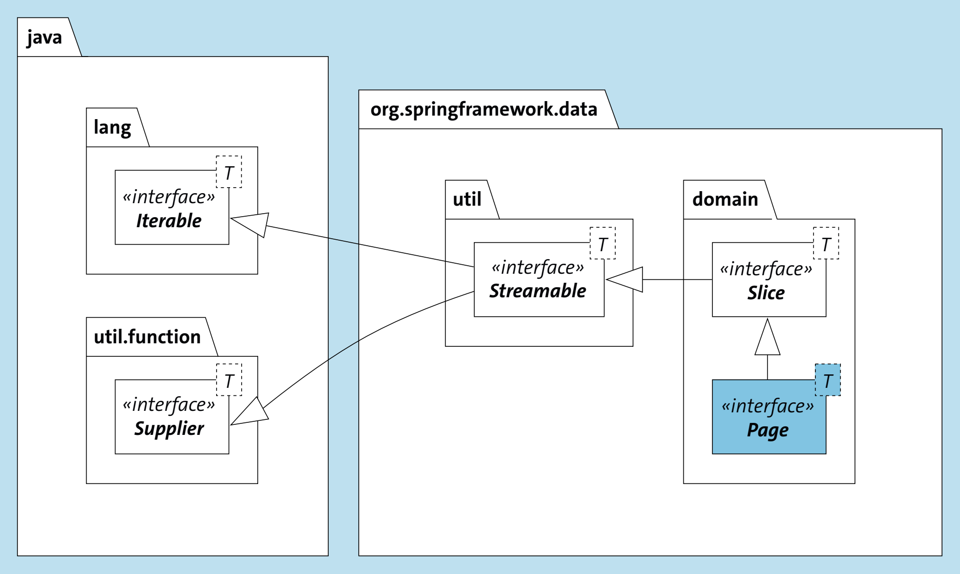

java
lang
T
«interface»
Iterable
util.function
T
«interface»
Supplier
org.springframework.data
util
T
«interface»
Streamable
domain
T
«interface»
Slice
T
«interface»
Page

Figure 7.9 UML Diagram of "Page" Type Relationships

The data type comes from Spring Data Commons and is used not only for Jakarta Persistence repositories but also for the other Spring Data projects.

The `Page` interface extends the `Slice` interface, which is a `Streamable`, and a `Streamable` is an `Iterable` and a `Supplier`. The `Page` data type doesn't have many methods:

- `static <T> Page<T> empty()`
- `static <T> Page<T> empty(Pageable pageable)`
- `long getTotalElements()`
- `int getTotalPages()`
- `<U> Page<U> map(Function<? super T,? extends U> converter)`

The `getTotalElements()` method returns the total number of all elements, not just the number of elements in the page. To get the total number, the Jakarta Persistence provider must execute a second `SELECT` statement to get this number with `COUNT`. `getTotalPages()`

returns the number of pages. The number of pages is calculated by `getTotalElements()` divided by the current page size.

In the `Slice` type, significantly more methods are added:

- `List<T> getContent()`
- `int getNumber()`
- `int getNumberOfElements()`
- `default Pageable getPageable()`
- `int getSize()`
- `Sort getSort()`
- `boolean hasContent()`, `boolean hasNext()`, `boolean hasPrevious()`, `boolean isFirst()`, `boolean isLast()`
- `Pageable nextPageable()`, `Pageable previousPageable()`
- `default Pageable nextOrLastPageable()`, `default Pageable previousOrFirstPageable()`
- `<U> Slice<U> map(Function<? super T,? extends U> converter)`

The `Slice` data type includes the crucial method `getContent()`, along with other methods that allow for requesting sizes, performing existence tests, and navigating through pages. The default methods `nextOrLastPageable()` and `previousOrFirstPageable()` are particularly useful for navigating because they prevent the user from getting stuck on the first or last page. In addition, the Slice interface includes a method `map(…)` that makes it easy to transform objects from the slice into new objects.

It's worth noting that the `Slice` interface extends the `Streamable` interface, which means that selected stream methods can be called directly on `Slice` objects. This data type was discussed earlier in Chapter 3, Section 3.11.3.

Task: Navigate through the Profiles

In the interactive application, you should be able to navigate through all profiles of the database. For this purpose, `RepositoryCommands` should be extended and store a state for the current page. The shell command `list` should display 10 profiles from the current page.

Two new shell commands are to be added:

- **`pp` (previous page)**
  Goes to the previous page and displays 10 profiles.
- **`np` (next page)**
  Goes to the next page and displays 10 profiles.

Proposed solution:

There are various methods to tackle this task. We can either remember the current page as a whole number (page 1, page 2, etc.) or as a `Page` object. However, if we remember the page as a number, it has two drawbacks:

- We have to repeatedly reload the data every time the list is called, which could be advantageous if we always want the latest data.
- We can't use the convenient slice navigation methods.

Therefore, we've opted for a different approach. We want to remember the `Page` and navigate with it. But when do we build the page? One option is to use an instance variable that remains unset until the first time the list or a `pp`/`np` command is called. Alternatively, we can build the `Page` in the constructor every time. However, neither option is perfect. `null` checks are error-prone, and we would rather not jump the gun. What we need is a container that fills up the first time it's accessed. Spring Data Commons provides such a class—`Lazy`—which we discussed previously in Chapter 3, Section 3.11.3.

The solution may look like the following listing.

```java
@ShellComponent
public class RepositoryCommands {
  private static final int PAGE_SIZE = 10;
  private final ProfileRepository profiles;
  private Lazy<Page<Profile>> currentPage;

  public RepositoryCommands( ProfileRepository profiles ) {
    this.profiles = profiles;
    currentPage =
      Lazy.of( () -> profiles.findAll(PageRequest.ofSize(PAGE_SIZE)) );
  }

  @ShellMethod( "Display all profiles" )
  public List<Profile> list() {
    return currentPage.get().getContent();
  }

  @ShellMethod( "Set current page to previous page, display the current page" )
  List<Profile> pp() {
    currentPage = currentPage.map(
                    page -> profiles.findAll( page.previousOrFirstPageable() ) );
    return list();
  }

  @ShellMethod( "Set current page to next page, display the current page" )
  List<Profile> np() {
    currentPage = currentPage.map(
                       page -> profiles.findAll( page.nextOrLastPageable() ) );
    return list();
  }
}
```

Listing 7.2 RepositoryCommands.java

The `Lazy` object is equipped with a `Supplier` that retrieves the data. The `Supplier` is only called when there is an access, either through the `list` command or through the `map(…)` methods. At this point, the data is fetched and stored in the `Lazy` object for future use.

The `map(…)` method is particularly useful as it allows for the creation of a new `Lazy` object from an existing `Lazy` object with transformed content. However, the `Supplier` isn't called until there is actually access to the data. This is guaranteed in this scenario as a call to `list()` will always follow `pp()`/`np()`.

7.4.5 Sort Paginated Pages

A `Pageable` object can be created with a sorting criterion, which will sort the dataset before pagination. This means that Spring doesn't just select the page and then sort it, but rather sorts the data before pagination. When constructing the sorting criteria, we can use the concepts learned in the previous section. Here are three examples:

```
Pageable sortedByManelength =
   PageRequest.of( 2, 10, Sort.by("manelength") );

Pageable sortedByManelengthDesc =
   PageRequest.of( 2, 10, Sort.by("manelength").descending() );

Pageable sortedByManelengthAndLastseen =
   PageRequest.of( 2, 10, Sort.by( Sort.Order.asc("manelength"),
                            Sort.Order.desc("lastseen") ) );
```

Of course, it's sorted first and then paginated.

Performance Considerations

The code for page navigation is written elegantly and quickly. Spring Data JPA implements pagination using `LIMIT` (including `FETCH`). As simple as this may be, it can have some costs in terms of performance. The problem is that sorting related to `LIMIT` and `OFFSET` can be costly. For nonsorted sets, this is still acceptable, but sorting makes `LIMIT` and `OFFSET` expensive. It's inefficient because the database is required to sort the data first, only to potentially discard a large portion of it later.

Internally, `PagingAndSortingRepository` uses a technique called *page-based pagination*. This uses `LIMIT` and `OFFSET`[13] in SQL, as we've seen. This is easy to program, but slow. Therefore, you should carefully consider whether `PagingAndSortingRepository` is really the right data type.

---

[13] Internally, `SimpleJpaRespository` sends a query with a pagination via the Criteria API; JPQL or SQL isn't built directly: *https://github.com/spring-projects/spring-data-jpa/blob/main/spring-data-jpa/src/main/java/org/springframework/data/jpa/repository/support/SimpleJpaRepository.java.*

There are other methods for pagination, including *keyset-based pagination* and *cursor-based pagination*. Offset- or keyset-based scrolling has been added since Spring Boot 3.1, but API support is only in its beginning stages.[14]

7.5 QueryByExampleExecutor *

In this section, we'll explore `QueryByExampleExecutor`, which is another fundamental type of Spring Data. Similar to `[List]CrudRepository` or the `JpaRepository`, this type provides predefined methods. `QueryByExampleExecutor` is designed to enable searching based on examples, similar to how people use examples to describe what they are searching for. For instance, if we're seeking a book that contains the word "Java" in its title, is in Spanish, and has been on the market for more than 10 years, or if we want all profiles that have been viewed in the past three months and are over 100 years old, we can define these queries as examples.

Thanks to an example, the library can generate a query for the database that returns results matching the *sample*. `QueryByExampleExecutor` is precisely designed for this purpose.

7.5.1 Sample

Let's assume we want to search for certain `Profile` instances. We first build a sample:

```
Profile p = new Profile( null, null,
                        /* manelength = */ 10,
                        /* gender = */     0,
                         null, null, null );
```

Almost all properties are set to `null` via the constructor; only the desired mane length should be `10` and the searched gender `0`.

---

[14] *https://docs.spring.io/spring-data/commons/docs/current/reference/html/#repositories.scrolling*

7.5.2 QueryByExampleExecutor

The data type that can return corresponding results for a sample is `QueryByExampleExecutor`.[15] Our repository interface must extend `QueryByExampleExecutor` to do this, and there are two options:

- **Variant 1**
  `JpaRepository` extends `QueryByExampleExecutor`, and the search methods are present:

  ```
  interface ProfileRepository extends JpaRepository<Profile, Long> { }
  ```

  If a repository extends `JpaRepository`—as we did—there is no need to include `QueryByExampleExecutor` again through the `extends` keyword.
- **Variant 2**
  If your own repository only extends `[List]CrudRepository`, `QueryByExampleExecutor` must also be extended because `[List]CrudRepository` doesn't extend `QueryByExampleExecutor`:

  ```
  interface ProfileRepository extends CrudRepository<Profile, Long>,
                                      QueryByExampleExecutor<Profile> { }
  ```

  Of course, it's allowed if the `ProfileRepository` also inherits only from `QueryByExampleExecutor`.

When using `QueryByExampleExecutor<Profile>`, it's worth noting that only the type argument `Profile` is relevant, while the ID type `Long` isn't significant.

QueryByExampleExecutor API

Here are all the methods provided by the `QueryByExampleExecutor` interface:

```
package org.springframework.data.repository.query;

import …

public interface QueryByExampleExecutor<T> {

  <S extends T> Optional<S> findOne(Example<S> example);
  <S extends T> Iterable<S> findAll(Example<S> example);
  <S extends T> Iterable<S> findAll(Example<S> example, Sort sort);
  <S extends T> Page<S> findAll(Example<S> example, Pageable pageable);
  <S extends T,R> R findBy(Example<S> example,
    Function<FluentQuery.FetchableFluentQuery<S>,R> queryFunction);
  <S extends T> long count(Example<S> example);
  <S extends T> boolean exists(Example<S> example);
}
```

---

[15] *https://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/repository/query/QueryByExampleExecutor.html*

In the parameter list, data type `Example`[16] can be found everywhere. An `Example` object encapsulates the sample.

7.5.3 Sample into the Example

The `Example` objects builds a factory method `of(…)`:

```
Profile p = new Profile( null, null, /* manelength */ 10, /* gender */ 0,
                         null, null, null );
Example<Profile> example = Example.of( p );
```

The `Example` object with the sample is passed to the methods of `QueryByExampleExecutor` in the next step:

```
List<Profile> result = profiles.findAll( example );
```

An `Example` object is immutable.

The execution generates SQL in the background with a `WHERE` clause that looks like this:

```
where p1_0.manelength=10 and p1_0.gender=0
```

Spring can extract information from a sample and create a `WHERE` clause based on it. For instance, you can read that it checks for a mane length of `10` and gender `0`. The queries are automatically concatenated with an ampersand, but this behavior can be customized using `ExampleMatcher`.

The `of(…)` method used to construct `Example` objects is overloaded, one of which allows for passing an `ExampleMatcher` (see Figure 7.10).

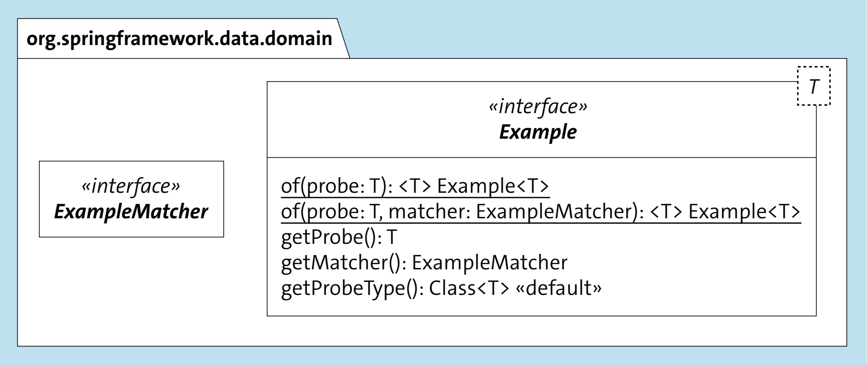

org.springframework.data.domain
«interface»
ExampleMatcher
«interface»
Example
T
of(probe: T): <T> Example<T>
of(probe: T, matcher: ExampleMatcher): <T> Example<T>
getProbe(): T
getMatcher(): ExampleMatcher
getProbeType(): Class<T> «default»

Figure 7.10 Operations in “Example”

---

[16] *https://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/domain/Example.html*

7.5.4 Build ExampleMatcher

The `ExampleMatcher` interface[17] provides static factory methods for building the instances:

- **`matchingAll()`**, alternatively **`matching()`**
  All values must match the sample, as in an AND logic operator. This is the default behavior.
- **`matchingAny()`**
  It's enough for individual values to match, as in an OR logic operator.

The following lines are synonymous:

```
var example = Example.of( p );
var example = Example.of( p, ExampleMatcher.matching() );
var example = Example.of( p, ExampleMatcher.matchingAll() );
```

The `ExampleMatcher` copies are immutable. A Fluent API can be used to further refine the states, so all methods return `ExampleMatcher` again.

OR Linking with ExampleMatcher

Suppose we want to know which profiles have either mane length `10` *or* gender equal to `0`:

```
Profile p = new Profile( null, null, 10, 0, null, null, null );
ExampleMatcher matcher = ExampleMatcher.matchingAny();
Example<Profile> example = Example.of( p, matcher );
```

For the `Example` object, `ExampleMatcher.matchingAny()` is passed for the OR query. The generated SQL will contain an `OR` instead of `AND`.

7.5.5 Ignore Properties

While `null` references are ignored in a query, primitive data types are always occupied and part of the query. However, it's important to ignore certain persistent attributes. Otherwise, how can we express that gender doesn't matter?

```
Profile p = new Profile( null, null, /* manelength */ 10, /* gender */ ????,
                         null, null, null );
```

It's not possible to pass `0`, use `null`, or ignore primitive values via the sample. `ExampleMatcher` provides a solution.

---

[17] *https://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/domain/ExampleMatcher.html*

ExampleMatcher + withIgnorePaths(…)

`ExampleMatcher` can be used to exclude certain properties from the SQL so that they no longer appear in the `WHERE` clause. If a persistent attribute isn't to be considered, we use `withIgnorePaths(String… ignoredPaths)`:

```
Profile p = new Profile( null, null, /* manelength */ 10, 0,
                         null, null, null );
var matcher = ExampleMatcher.matching().withIgnorePaths( "gender" );
Example<Profile> example = Example.of( p, matcher );
List<Profile> result = profiles.findAll( example );
```

In the profile sample, `gender` is set to `0`, but basically the value doesn't matter and is ignored.

> **Note**
>
> By default, reference variables with `null` are ignored. Of course, it may be that a query explicitly wants to check for `null`. This can be changed with `IncludeNullValues()`.

With `ExampleMatcher`, you can also do a little more.

7.5.6 Set String Comparison Techniques

Strings are compared completely by default, so they must match character for character. This can be controlled because it must also be possible to ask if a substring occurs or if case isn't important.

The property can be controlled with a `StringMatcher` passed in the `ExampleMatcher` method `withStringMatcher(StringMatcher)`. The `StringMatcher` parameter type[18] is an enumeration type with the constants `EXACT`, `STARTING`, `ENDING`, `CONTAINING`, and `REGEX`; the default is `DEFAULT`.

The `withStringMatcher(…)` method sets the behavior for all string properties. We'll see in a moment how this can be controlled individually for the different persistent attributes.

Another method is `withIgnoreCase(String... propertyPaths)`; it helps to ignore case. It passes the names of the persistent attributes, and the method returns an `ExampleMatcher` that allows a Fluent API. Let's take a look at an example for this:

```
final int IGNORE = 0;
Profile p = new Profile( "st", null, IGNORE, IGNORE, null, null, null );

ExampleMatcher matcher = ExampleMatcher.matching()
```

---

[18] *https://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/domain/ExampleMatcher.StringMatcher.html*

```
    .withIgnorePaths( "manelength", "gender" )
    .withStringMatcher( ExampleMatcher.StringMatcher.CONTAINING )
    .withIgnoreCase();

Example<Profile> example = Example.of( p, matcher );
```

The goal is to get all profiles that contain the nickname `"st"`. The persistent attributes `manelength` and `gender` should both be completely ignored. To make this a bit more readable, there is a constant in the program that could be assigned any value.

7.5.7 Set Individual Rules with GenericPropertyMatcher

The configuration via `withStringMatcher(…)` applies to all persistent string attributes in the `ExampleMatcher` object. But there is also the possibility to select the setting locally for a specific persistent attribute. For this, method `withMatcher(String propertyPath, GenericPropertyMatcher genericPropertyMatcher)` is used. The method is first passed the name of the persistent attribute, and then `GenericPropertyMatcher` configures the details. `GenericPropertyMatcher`[19] is a class with static factory methods that are symmetric to the constants in `StringMatcher`: `ignoreCase()`, `caseSensitive()`, `contains()`, `endsWith()`, `startsWith()`, `exact()`, and `regex()`.

In this example, ignore `manelength` and `gender`, and the sample should contain a substring at `nickname`:

```
ExampleMatcher matcher = ExampleMatcher.matching()
   .withIgnorePaths( "manelength", "gender" )
   .withMatcher( "nickname",
                 GenericPropertyMatchers.contains().ignoreCase() );
```

In the example, both solutions work. However, this `GenericPropertyMatcher` has a feature that `ExampleMacher` with its global configuration doesn't have: the assignment in the sample can be transformed before going to the database.

7.5.8 PropertyValueTransformer

A change of the parameters takes over a `PropertyValueTransformer`.[20] The use of this type requires several steps.

Let's use the `PropertyValueTransformer` for an example and assume that the nickname may contain whitespace in the sample, but that all whitespace should be removed before the string is passed as a query parameter to the database.

---

[19] *https://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/domain/ExampleMatcher.GenericPropertyMatcher.html*

[20] *https://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/domain/ExampleMatcher.PropertyValueTransformer.html*

```
PropertyValueTransformer removeAllSpaces =
    optional -> optional.filter( String.class::isInstance )
                        .map( String.class::cast )
                        .map( s -> s.replaceAll( "\\s+", "" ) );
```

A `PropertyValueTransformer` is a special `Function`; the declaration is as follows:

```
public static interface ExampleMatcher.PropertyValueTransformer
extends Function<Optional<Object>,Optional<Object>> { }
```

The type is nested in `ExampleMatcher`.

The type signature indicates that the `PropertyValueTransformer` converts an `Optional` into another `Optional`. The implementation first checks if the `Optional` contains a `String`. If so, the type `java.lang.Object` is converted into `java.lang.String`, and all whitespace is removed from the string.

You can pass `MatcherConfigurer<GenericPropertyMatcher>`[21] to `withMatcher(…)` and use it to set a property transformer:

```
ExampleMatcher matcher = ExampleMatcher.matching()
    .withIgnorePaths( "manelength", "gender" )
    .withMatcher( "nickname", m -> m.transform( removeAllSpaces )
                                    .contains().ignoreCase() );
final int IGNORE = 0;
var p = new Profile("Fillmore FAT", null, IGNORE, IGNORE, null, null, null);
Example<Profile> example = Example.of( p, matcher );
```

This allows a property to be modified before it's used in a query.

Evaluation of Query By Example

The `QueryByExampleExecutor` interface using the Query By Example (QBE) language provides both advantages and disadvantages. One of its main advantages is that it's resilient to refactoring, as the queries are relatively free of string literals (except for persistent attributes that are to be ignored). Therefore, if there are changes to types or identifiers within the sample, the query will still likely work as before. Moreover, `QueryByExampleExecutor` is supported by various data stores, such as relational databases and MongoDB, making it a versatile option.

As mentioned, there are also several disadvantages to using `QueryByExampleExecutor`. Although different Spring Data implementations support it in principle, not all of its API features may be supported, such as using `REGEX` with `StringMatcher`. Additionally, range specifications aren't possible, meaning that we can't, for example, check which profiles have a mane length between `1` and `100`, which is a limitation for certain use cases. Furthermore, nested queries of the form “Mane length is X or (birthday on YY and gender = Z)” are currently not supported. Because these disadvantages outweigh the advantages, `QueryByExampleExecutor` doesn’t play a significant role in practice in its current form.

---

[21] *https://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/domain/ExampleMatcher.MatcherConfigurer.html*

7.6 Formulate Your Own Queries with @Query

`[List]CrudRepository` offers useful methods such as retrieving all entities, storing multiple entity beans, and deleting entities by IDs, which surpasses the capabilities of `EntityManager`. Nevertheless, these pre-implemented methods fall short in practical use. To provide more specific data queries, a complete repository should include methods that filter profiles by certain criteria or unicorns by email addresses. These requirements can’t be met by `[List]CrudRepository`. However, it’s not necessary to revert to `EntityManager` because it’s possible to augment your own repository with JPQL or native queries.

7.6.1 @Query Annotation

Spring Data provides the `@Query`[22] annotation, which can be used to define new methods in the repository interface. By adding the `@Query` annotation, the database query can be written in the native language of the database, such as JPQL, SQL, or MongoDB queries. However, unlike `[List]CrudRepository` and `[List]PagingAndSortingRepository`, methods with the `@Query` annotation are dependent on the specific data store. Therefore, if the database management system is changed, the queries may need to be adapted accordingly.

For instance, let’s consider the scenario where all profiles interested in other profiles of any gender need to be retrieved. In the database, this is represented by `NULL` in the `attractedToGender` column, indicating that the profile is open to both genders or hasn’t specified a preference in column `1` or `2`. The following code snippet demonstrates how this can be achieved:

```
public interface ProfileRepository extends JpaRepository<Profile, Long> {
  @Query( "SELECT p FROM Profile p WHERE p.attractedToGender IS NULL" )
  List<Profile> findBiUnicorns();
}
```

The new method goes into `ProfileRepository` and is annotated with `@Query`. The annotation is from the `org.springframework.data.jpa.repository` package. Other Spring Data projects use the same annotation, but from different packages!

---

[22] *https://docs.spring.io/spring-data/jpa/docs/current/api/org/springframework/data/jpa/repository/Query.html*

The JPQL query is formulated in a typical manner, and the method can have different return types. In this case, because multiple profiles can be returned, using a `List` is appropriate, although a `Collection` or `Stream` could also be used. In situations where the query results in only one profile, the return type can be a single `Profile` instance. On the other hand, if there is no result, then the return would be `null`. However, it's more common to use `Optional<Profile>` as the return type in API design, to handle the case where there are no results.

@Query Annotation for Parameterized Methods

Parameters can be added to the `@Query` method by making the method parameterized. There are two options: positional parameters or named parameters. Let's take a look at the same query using both options. The query is intended to list profiles that were viewed after a certain date.

For the first option, here are the *position parameters*:

```
@Query( "SELECT p FROM Profile p WHERE p.lastseen > ?1" )
List<Profile> findProfilesLastSeenAfter( LocalDateTime timestamp );
```

The JPQL uses a `WHERE` clause to restrict the results in which `profilesLastSeen` must be greater than the passed value. The symbol `?` is the placeholder for the prepared statement and is filled with whatever was passed to the method. The position parameter is also called the *index parameter*.

An alternative notation is to refer to an identifier with a colon, which is called *named parameter*:

```
@Query( "SELECT p FROM Profile p WHERE p.lastseen > :lastseen" )
List<Profile>
   findProfilesLastSeenAfter( LocalDateTime lastseen );
// findProfilesLastSeenAfter( @Param( "lastseen" ) LocalDateTime timestamp );
```

In JPQL, the identifier preceded by a colon is named after the parameter variable `lastseen`. If the parameter variable needs to be named differently, such as `timestamp`, the `@Param` annotation must be used to specify the named parameter in the JPQL string.

Task: Introduce New @Query-Annotated Methods

If you want to practice, you can implement the following three methods with an appropriate `@Query`:

- `findProfileByNickname(String name)`
- `findProfilesByContainingName(String name)`
- `findProfilesByManelengthBetween(short min, short max)`

Proposed solution:

```
@Query( "SELECT p FROM Profile p WHERE p.nickname = :name" )
Optional<Profile> findProfileByNickname( String name );

@Query( "SELECT p FROM Profile p WHERE p.nickname LIKE %:name%" )
List<Profile> findProfilesByContainingName( String name );

@Query( "SELECT p FROM Profile p WHERE p.manelength BETWEEN :min AND :max" )
List<Profile> findProfilesByManelengthBetween( short min, short max );
```

7.6.2 Modifying @Query Operations with @Modifying

In JPQL, an expression can contain an `UPDATE` or `DELETE` statement in addition to `SELECT`. As we saw in Chapter 6, Section 6.8.9, this is typically used for batch updates. When executing `UPDATE` or `DELETE` queries using JPQL strings via `EntityManager`, a different `EntityManager` method must be used compared to `SELECT` queries. Specifically, `executeUpdate()` is used instead of the `getResultList()`, `getResultStream()`, and `getSingleResult()` methods, respectively. However, because Spring Data JPA can't determine from the string whether it's a `SELECT`, `UPDATE`, or `DELETE` query, the `@Modifying` annotation must be added to the `@Query` method for modifying operations.

Here's an example that demonstrates how to update the `lastseen` timestamp for a profile with a specific ID:

```
@Modifying
@Query( "UPDATE Profile p SET p.lastseen =:lastseen WHERE p.id = :id" )
int updateLastSeen( long id, LocalDateTime lastseen );
```

The example resorts to two named parameters. The profile to be updated is identified by `id`, and `lastseen` is again passed as `LocalDateTime`.

If you call `@Modifying` operations from the outside, they are changes and must take place in a transactional context. Schematically, it looks like this:

```
@Transactional
@Override public void run() {
  long id = …
  profiles.updateLastSeen( 1, LocalDateTime.now() );
}
```

7.6.3 Fill IN Parameter by Array/Varg/Collection

Both JPQL and SQL provide the ability to use the `IN` operator to create a concise OR query that checks whether a persistent attribute has certain values in the database. For instance, if we want to retrieve profiles that express interest in specific genders:

```
@Query( "SELECT p FROM Profile p WHERE p.attractedToGender IN :genders" )
List<Profile>    findProfilesAttractedToGender( Byte... genders );
// List<Profile> findProfilesAttractedToGender( List<Byte> genders );
// List<Profile> findProfilesAttractedToGender( Byte[] genders );
```

It's possible to pass values to the `IN` operator of a query from the Java side using a vararg, a list, or an array. In addition to `IN`, there is also a negation operator: `NOT IN`.

7.6.4 @Query with JPQL Projection

The methods annotated with `@Query` have returned either one or more entity beans. But it doesn't have to be that way because we've already seen in Chapter 6, Section 6.6.9, that a JPQL expression can also contain a projection. Then the result is no longer an entity bean, but perhaps an integer value, a string, or an associative store with key-value pairs.

For example, the `SELECT` statement should not return a `Profile` entity bean, but only the ID of a profile and the nickname:

```
@Query(
 "SELECT p.id as id,p.nickname as nickname FROM Profile p WHERE id=:id"
)
Map<String, Object> findSimplifiedProfile( long id );
```

Method `findSimplifiedProfile(…)` returns a `Map` with key-value pairs, where the keys in our case are the `id` and the `nickname`, and the associated values are `long` and `String`. Because the ID can only occur once, one `Map` is sufficient for the "row."

If more values are returned, the result is a list of small associative stores. For example, we want to select all profiles and also project them to `ID` and `nickname`:

```
@Query("SELECT p.id as id,p.nickname as nickname FROM Profile p")
List<Map<String, Object>> findAllSimplifiedProfiles();
```

If you call the method, you get a `List` of `Map`s. So, you could log the contents:

```
Map<String, Object> map = profiles.findSimplifiedProfile( 1 );
log.info( "id={}, nickname={}", map.get( "id" ),
                                map.get( "nickname" ) );

List<Map<String, Object>> maps = profiles.findAllSimplifiedProfiles();
maps.forEach( map -> log.info( "id={}, nickname={}",
                               map.get( "id" ), map.get( "nickname" ) ) );
```

7.6.5 Sort and Pageable Parameters

`Sort` and `Pageable` are important data types of Spring Data. `Sort` represents an order criterion, and `Pageable` describes a page. `Sort` and `Pageable` objects can additionally be passed to `@Query` methods. In this case Spring Data JPA will automatically integrate this `Sort` and `Pageable`, respectively, into the query. The advantage is that the ordering criterion is flexible, and there is no need to have different `@Query` methods with hard-coded criteria. For example, we want to find all profiles that were seen after a certain time:

```
@Query( "SELECT p FROM Profile p WHERE p.lastseen > :lastseen" )
List<Profile> findProfilesLastSeenAfter( LocalDateTime lastseen,
                                         Sort sort );
```

Unlike the queries before, the JPQL expression doesn't fall back on the parameter. Spring Data recognizes that `Sort` is a special data type and adds the sort criterion to the JPQL string.

`Pageable` is used to query pages and sections. We can add a variant to the `findProfilesLastSeenAfter(…)` method:

```
@Query( "SELECT p FROM Profile p WHERE p.lastseen > :lastseen" )
// with getTotalElements()
Page<Profile>  findProfilesLastSeenAfter( LocalDateTime lastseen,
                                          Pageable p );
```

The difference is that we pass `Pageable` along with `lastseen`, which, of course, can itself receive a sort again.

If `Pageable` is passed, different return types are allowed, namely `Page`, `Slice`, or a collection type such as `List`. If the return is a `Page` object, the number of total elements is important and must be queried via a separate `SELECT COUNT`. Instead of one query, two SQL queries are required for the `Page` return type. This is no longer the case for the `Slice` and `List` data types.

```
// does not know total
Slice<Profile> findProfilesLastSeenAfter(LocalDateTime lastseen, Pageable p);
// no more navigation
List<Profile>  findProfilesLastSeenAfter(LocalDateTime lastseen, Pageable p);
```

If we use a `Slice`, there is still the possibility of navigation, but the `Slice` lacks the information about the total number of elements. If we abstract even further, that is, down to the `List` data type, it's only the data of the query itself.

Note

If `Sort` or `Pageable` occur in the parameter list, callers must not pass `null`. If no sorting or pagination details are desired or known, `Sort.unsorted()` or `Pageable.unpaged()` can be used.

JpaSort

The `Sort` data type operates exclusively on persistent attributes that are known. Other sorting criteria aren't feasible with this type. Nonetheless, there is a need to sort based on other criteria, which is why Spring Data introduces the `JpaSort` data type, a subclass of `Sort`. The `JpaSort` objects can be created using the static factory method `JpaSort.unsafe(…)`. This approach can be applied to use a function as a sorting criterion, as demonstrated in the following example:

```
var result = profiles.findProfilesLastSeenAfter(
                 LocalDateTime.now().minusYears( 10 ),
                 JpaSort.unsafe( "LENGTH(nickname)" ) );
```

The sort criterion is the length of the nicknames. The `Sort` object could not express something like that.

Note

The `JpaSort` class extends `Sort` and inherits the `of(…)` method, so `JpaSort.of(…)` also returns only one `Sort` object. It's important to use the `unsafe(…)` method.

7.6.6 Add New Query Methods

When using an `@Query` method in Spring Data, it's essential that the number and types of parameters align with the placeholders used in the JPQL query. However, this approach may not be flexible enough, especially when aiming to provide an optimal "caller experience" for the client. If there's a need to introduce new repository methods that involve preprocessing of the parameters, there are three possible options to consider:

- Add your own default methods.
- Introduce Spring Expression Language (SpEL) expressions in JPQL.
- Realize a fragment interface with your own implementation.

We'll look at the first two options now and the fragment interface later in Section 7.11.3.

Default Methods

We formulated our first `@Query` method like this:

```
@Query( "SELECT p FROM Profile p WHERE p.lastseen > ?1" )
List<Profile> findProfilesLastSeenAfter( LocalDateTime timestamp );
```

What if the query should be possible not only by a precise date, but by a year, with the starting point then being the beginning of the year? Then, a new default method can be introduced that takes a `Year` object, converts it to a `LocalDateTime`, and uses it to call its own `findProfilesLastSeenAfter(LocalDateTime)` method:

```
default List<Profile> findProfilesLastSeenAfter( Year lastseen ) {
  return findProfilesLastSeenAfter( lastseen.atDay( 1 ).atStartOfDay() );
}
```

Default methods can realize complex operations, but there are two disadvantages:

- They only complement methods, but don't hide other methods.
- Because they are in an interface, they can't rely on any other Spring-managed beans.

7.6.7 Queries with Spring Expression Language Expressions

JPQL strings in a `@Query` can contain SpEL expressions. This allows queries to include other Spring-managed beans and transform parameters before inserting them into the JPQL query.

Let's come back to the question of which profiles have logged in after a certain year. With SpEL, our previous `default` method isn't necessary:

```
@Query( """
        SELECT p FROM Profile p
        WHERE p.lastseen > ?#{#lastseen.atDay(1).atStartOfDay()}""" )
List<Profile> findProfilesLastSeenAfter( Year lastseen );
```

When using SpEL expressions in Spring Data JPA, they need to be declared in the `@Query` string to be recognized and transformed before being sent to `EntityManager`. SpEL expressions are identified by a colon or a question mark at the beginning, as shown in our example using a question mark.

The parameters can be referenced using either their name or index enclosed in square brackets. As a result, there are four different notations to choose from. The alternative notations are as follows:

- … `WHERE p.lastseen > ?#{[0].atDay(1).atStartOfDay()}`
- … `WHERE p.lastseen > ?:{[0].atDay(1).atStartOfDay()}`
- … `WHERE p.lastseen > :#{#lastseen.atDay(1).atStartOfDay()}`

As usual, the following applies in life: no matter how you do it, the main thing is to be consistent.

SpEL has another variable in the namespace, namely `{#entityName}`. This can be used to dynamically query the name of the entity bean. The advantage is that you wouldn't have to change the entity name when refactoring.

The selection between the question mark and the colon isn't arbitrary, as we've observed that the question mark denotes an index parameter, while the colon represents a named parameter. Additionally, these two symbols have a specific function within the `@Query` context.

7.6.8 Using the @NamedQuery of an Entity Bean

It's worth noting that `@Query` definitions are usually added to a method, but that's not always the case. In Chapter 6, Section 6.6.10, we learned about *named queries*, which are JPQL expressions linked to the `@Entity` class or defined in the *orm.xml* XML file. The primary benefit of named queries is that they keep JPQL expressions local to the entity and prevent them from spreading throughout the source code. This means that if the entity is modified, such as when something is renamed, it's easy to locate the JPQL string at the top of the class.

To demonstrate, let's add a `@NamedQuery` to the entity bean:

```
@Entity
@NamedQuery( name  = "Profile.findByNickname",
             query = "SELECT p FROM Profile p WHERE p.nickname = :nickname" )
public class Profile …
```

Here's the approach: If a repository has a method named `findByNickname(…)`, there's no need to use the `@Query` annotation, as Spring Data JPA will search for a named query. However, the name of the named query must follow a specific structure: it must begin with the entity's name (in our example, `Profile`), followed by a dot as a separator, and then the method's name. Therefore, the following would be valid in `ProfileRepository`:

```
public interface ProfileRepository extends JpaRepository<Profile, Long> {
  Optional<Profile> findByNickname( String nickname );
}
```

Note

If you rename this method, the named query must be renamed as well; otherwise, this won't work due to *derived query methods* (more about this follows in Section 7.8) and there will be strange effects. Named queries are good in themselves, but with mapping to methods, it's a bit shaky.

7.6.9 @Query Annotation with Native SQL

Until now, we've exclusively used JPQL for our queries. However, `EntityManager` also permits the use of Native SQL, which is also possible with Spring Data JPA. When `@Query` involves Native SQL, Spring can't discern whether it's JPQL or Native SQL in the `@Query` string. To distinguish between the two, an annotation attribute named `nativeQuery` must be enabled and set to `true`, as shown in the following example:

```
@Query( value       = "SELECT * FROM Profile WHERE manelength > ?1",
        nativeQuery = true )
List<Profile> findProfilesWithManelengthGreaterThan( short minManelength );
```

The native query in this case selects all columns from the `Profile` table and uses a `WHERE` clause to filter profiles based on mane length exceeding the bind variable. Some Jakarta Persistence providers may allow the use of named parameters (`:name`) within this native query, but this feature is dependent on the provider's implementation and not standardized. To ensure standardization, index parameters should be used instead. These are designated using a question mark, with the index starting from `1`.

It's possible to make modifications with Native SQL queries, but `@Modifying` must be enabled again, as shown in Section 7.6.2.

Pagination with Native Queries

We've just seen `findProfilesWithManelengthGreaterThan(…)` as a native query. Methods with Native SQL statements can also have a `Pageable` object passed to them, as we saw in JPQL queries in Section 7.6.5:

```
@Query( value = "SELECT * FROM Profile WHERE manelength > ?1",
        nativeQuery = true )
Page<Profile> findProfilesWithManelengthGreaterThan( short minManelength,
                                                     Pageable pageable );
```

When the return value is a `Page` object, two SQL queries are required—one for retrieving the data and another for the total count of elements. When certain constraints are applied, the total number of elements is expected to decrease. Therefore, a simple `SELECT COUNT(*)` statement isn't accurate, and the `WHERE` clause must also be considered when retrieving the total count. Without the `WHERE` clause, the count would generally be too high.

Spring Data JPA has the capability to automatically set the `WHERE` clause and generate a statement such as the following:

```
SELECT COUNT(*) FROM profile WHERE manelength > ?
```

It may be that a native query with the `WHERE` clause is so complicated that Spring can't generate a correct `SELECT COUNT`. Therefore, a `SELECT COUNT` can be manually. For example, it looks like this:

```
@Query( value = "…",
        countQuery = "SELECT count(*) FROM … WHERE …",
        nativeQuery = true )
```

It's more than useful to check the generated SQL from Spring Data JPA to see if the query is correct.

Sorting Native Queries

A `Pageable` object can include a `Sort` object to specify a sorting order. It may seem odd that `Sort` objects can't be passed directly into native `@Query` methods, but it's possible to include the sorting criterion in the query through the `Pageable` object.

As an example, let's take a look at the method declaration for `findProfilesWithManelengthGreaterThan(…)` and add a sorting criterion to the `Pageable` object when invoking the method:

```
var page = profiles.findProfilesWithManelengthGreaterThan(
                        (short) 4,
                        PageRequest.of( 0, 10, Sort.by( "nickname" ) ) );
```

The generated SQL statement contains this sort criterion and also `LIMIT` and potentially `OFFSET` via this `PageRequest`:

```
SELECT * FROM Profile WHERE manelength > ? order by nickname asc limit ?
select count(*) FROM profile WHERE manelength > ?
```

Spring Data JPA appends an `ORDER BY` to our SQL. This gets a bit more interesting because if we already have an `ORDER BY`, Spring Data JPA needs to take that into account. Let's say the query is as follows:

```
@Query( value = """
                SELECT * FROM Profile
                WHERE manelength > ?1 ORDER BY manelength DESC",
        nativeQuery = true )
Page<Profile> findProfilesWithManelengthGreaterThan( short minManelength,
                                                     Pageable pageable );
```

Then, the following SQL would be generated:

```
SELECT *
FROM Profile
WHERE manelength > ?
ORDER BY manelength DESC, nickname asc
limit ?
```

In this case, of course, it's an afterthought sort criterion, but it's still impressive that Spring Data JPA can recognize and rewrite SQL statements.[23]

7.7 Stored Procedures *

A stored procedure is a special feature of some databases that allows several SQL statements to be combined into a kind of subroutine. This has several advantages:

---

[23] If you would like to learn the details for JPQL expressions, the class `QueryUtils` is explained here: *https://docs.spring.io/spring-data/jpa/docs/current/api/org/springframework/data/jpa/repository/query/QueryUtils.html*

- The client doesn't have to realize several SQL queries, but only start the stored procedure. This reduces communication, and ultimately it's faster to make only one call to the database instead of going to the server several times and realizing different calls one after another.
- There are parameters and returns, which forms an API for business data. Stored procedures are thus a kind of API for communication with the database. Thus, details (e.g., names of tables or columns) can be hidden in the stored procedures and aren't visible to outsiders.
- This abstraction increases security because if only stored procedures are visible to the outside world and everything else is invisible, then you no longer come into contact with the raw data.

In practice, you often have both: selected tables and stored procedures. But if you wanted to take it to the extreme, you could communicate with the database exclusively via stored procedures.

Although stored procedures have their advantages, there are also disadvantages:

- The main problem is that stored procedures aren't part of the SQL standard. So far, standardization has never been pushed, which means that stored procedures aren't portable. Often, database management systems implement many extensions, so stored procedures read almost like imperatively programmed programs: there are variables, case distinctions, loops, and so on. The best-known dialects are *PL/SQL* (from Oracle) or *Transact SQL* (T-SQL; from Microsoft). Some database management systems don't have stored procedures.
- The development of stored procedures is also more difficult. If these exist exclusively on the database side, developers are dependent on the tools of the manufacturers and their debugging options. For example, the ability to set breakpoints in scripts varies.
- Because stored procedures are on the database side, there is a problem if they are deleted or if you want to maintain a new version in version management: they aren't necessarily part of your program code. It's unfortunate if the stored procedure is developed over several generations only within the database, and then outside the database, this script appears nowhere at all. That's why it's also important that installation and migration scripts are managed outside the database to set up the database in an automated way. This also helps with test cases that are only possible in cooperation with a database. Here, tools like *Flyway* help immensely—we'll look at this in Section 7.15.

7.7.1 Define a Stored Procedure in H2

Stored procedures in H2 are implemented as Java programs with special methods, unlike other RDBMSs where SQL statements are combined into a script. These Java methods can take external arguments and return tables as `ResultSet`, simple lists, or primitive elements. In the case of H2, a virtual table with one column is required to generate random names. The stored procedure for this is named `GET_RANDOM_NAMES`, and here's an example implementation:

```
DROP ALIAS IF EXISTS GET_RANDOM_NAMES;
CREATE ALIAS GET_RANDOM_NAMES AS $$
import java.util.concurrent.ThreadLocalRandom;
@CODE
java.sql.ResultSet getRandomNames( java.sql.Connection __, int size ) {
  org.h2.tools.SimpleResultSet rs = new org.h2.tools.SimpleResultSet();
  rs.addColumn( "RANDOM_NAME", java.sql.Types.VARCHAR, 255, 0 );
  for ( int i = 0; i < size; i++ )
    rs.addRow( getName() );
  return rs;
}
static String getName() {
  int size = ThreadLocalRandom.current().nextInt( 6, 16 );
  StringBuilder newName = new StringBuilder();
  for ( int i = 0; i < size; i++ )
    newName.append( i % 2 == 0 ?
      "aeiou".charAt( ThreadLocalRandom.current().nextInt( 5 ) ) :
      "bcdfghklmnprstvwyz".charAt(ThreadLocalRandom.current().nextInt(18)) );
  newName.setCharAt( 0, Character.toUpperCase( newName.charAt( 0 ) ) );
  return new String( newName );
}
$$;
CALL GET_RANDOM_NAMES (10);
```

Here are the details:

- Using `DROP ALIAS` before creating a stored procedure isn't strictly necessary, but it ensures that the procedure is deleted if it already exists, as stored procedures aren't automatically overwritten and would cause an error.
- Notation `CREATE ALIAS` is specific for H2 because that defines the function. After `$$`, the actual Java code begins. First, the `import` declarations come, and then after the separator `@CODE`, the Java methods come, which are the body of the class.
- Two methods are declared: `getRandomNames(…)` and utility method `getName()`. The helper method returns fantasy names, where consonants are simply between vowels; the string length is random.
- Method `getRandomNames(…)` takes `java.sql.Connection` as a parameter, which isn't required for our purposes but is included to demonstrate that the SQL connection can be used to connect to your own database. The `size` parameter determines the number of generated names, which are returned via a virtual table that creates a

special `ResultSet`. The `SimpleResultSet` is a data type in H2 and is a writable `ResultSet`. It creates a table with `size` as many random names, with each name in its row.

- `CALL GET_RANDOM_NAMES (10)` is an example of a call to this stored procedure.

Now, the call of the stored procedure should not be done via SQL, but via a Spring program. There are two ways to do this, which we'll look at in the following sections.

7.7.2 Calling a Stored Procedure via a Native Query

One of the ways uses the `@Query annotation` with a native query:

```
@Query( value       = "CALL GET_RANDOM_NAMES(?1)",
        nativeQuery = true )
List<String> getRandomNames( int size );
```

The SQL string is almost identical as before, only here we have a pass with `?1`. Repository method `getRandomNames(…)` can be passed an argument. This fills the bind variable of the prepared statement. In our case, the return is a list of strings. If the stored procedure returns the columns of an entity bean, the return can also be transferred to an entity bean. The use of the `CALL` keyword is specific to the RDBMS because it's Native SQL and not JPQL. Therefore, there is a second possibility.

7.7.3 Call a Stored Procedure with @Procedure

A repository method can be annotated with `@Procedure`[24] so that the native query can be omitted:

```
@Procedure
List<String> GET_RANDOM_NAMES( int size );
```

The Java method is named the same as the stored procedure. Arguments passed to the method are passed when the stored procedure is called. The call is internal and hidden by Spring. Moreover, as before, the return is a list of strings.

Because the method name violates the JavaBean convention and might collide with another method or a Java keyword and because there might be a problem when renaming the stored procedure, `@Procedure` lets you specify the name of the stored procedure:

```
@Procedure( "GET_RANDOM_NAMES" )
List<String> getRandomNames( int size );
```

This allows the name to be different on the Java side and makes the code independent of the stored procedure's name.

---

[24] *https://docs.spring.io/spring-data/jpa/docs/current/api/org/springframework/data/jpa/repository/query/Procedure.html*

Note: Call @Procedure Transactionally

The `@Procedure` method must be called transactionally, even if it doesn't write anything:

```
@Transactional( readOnly = true )
public void run() {
  List<String> names = profiles.getRandomNames( 10 );
  …
}
```

With the `readOnly = true` flag, the driver or the database can execute the query optimized because the transaction doesn't change anything and only reads.

7.8 Derived Query Methods

Previously, you've become familiar with several `*Repository` interfaces provided by Spring Data, which declare a set of methods for data access. You've also seen how to add additional methods to our own repository interfaces and annotate them with `@Query` to define custom queries using JPQL or SQL.

However, not all queries require the use of `@Query` annotations. Spring Data provides the ability to generate queries automatically based on method names via the derived query method, as mentioned earlier. By choosing method names in a certain way, Spring Data can infer the query logic and generate the necessary SQL or JPQL on the fly.

7.8.1 Individual CRUD Operations via Method Names

Let's take a look at what derived query methods look like using two repository interfaces:

```
public interface ProfileRepository extends JpaRepository<Profile, Long> {
  List<Profile> findByBirthdate( LocalDate birthdate );
  List<Profile> findByDescriptionContaining( String text );
  long          countByManelength( short manelength );
}

public interface UnicornRepository extends JpaRepository<Unicorn, Long> {
   long          deleteByEmail( String email );
}
```

Different things stand out. The `@Query` annotation is missing, and there is no named query either. The method names have a special structure: In front there is a verb that formulates the goal of the query, followed with a separator "`by`", and then you access persistent attributes with a special method fragment, for example:

- `findByBirthdate(…)`
  Which birthdate? The one that is passed via the parameter list.
- `findByDescriptionContaining(…)`
  What `description`? It comes from the parameter.
- `countByManelength(…)`
  Which `manelength` should be counted? That comes from the parameter!

A `Unicorn` has an email address, and with method `deleteByEmail(…)` in `UnicornRepository`, the `Unicorn` with the email address can be deleted.

Derived query methods are a feature in Spring Data that allow semantic method names to be internally translated into a query without the need for the `@Query` annotation. However, for this to work, the method names must adhere to a specific structure; otherwise, an error will occur during the startup of the Spring Boot application. Fortunately, many modern development environments provide checks to ensure that the method names match the persistent attributes.

7.8.2 Structure of the Derived Query Methods

We need to take a closer look at the query builder mechanism. A railroad diagram[25] for the construction plan of the method names is shown in Figure 7.11.

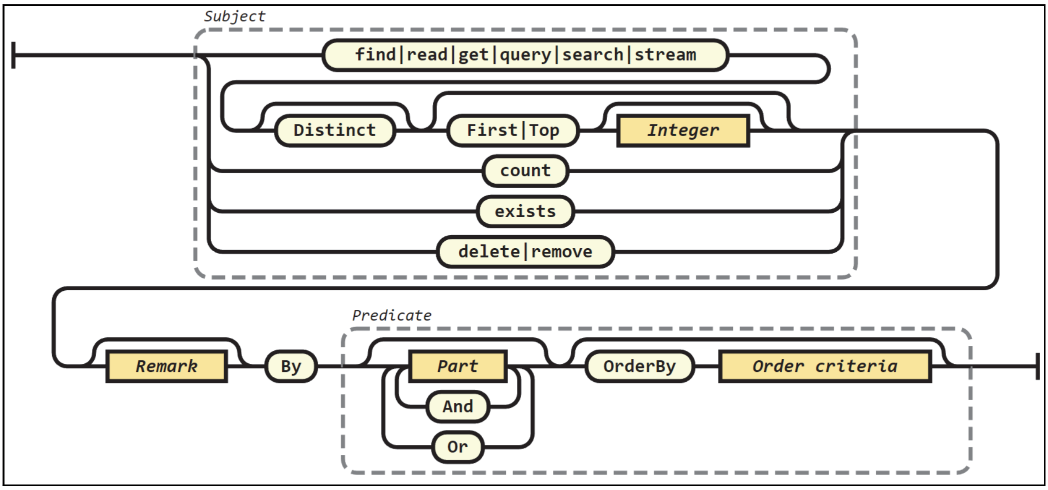


Figure 7.11 Structure of the Derived Query Methods

A method name encodes two pieces of information: the *subject* and the *predicate*. The subject is the query, that is, what it's about. Prefixes such as `find`, `read`, `get`, `query`, `search`, or `stream` direct a search query; all terms separated by a vertical bar symbolize synonyms. Thereafter, we can optionally use `Distinct`. We can also append `First` or `Top`, which returns the first element, or put an integer after `First` or `Top` for a given number of first elements.

---

[25] This diagram is derived from the source code of the method name parser: *https://github.com/spring-projects/spring-data-commons/blob/main/src/main/java/org/springframework/data/repository/query/parser/PartTree.java*.

If the result should not be entities or properties, but the number of elements is requested, `count` is used. The prefix `exists`, on the other hand, asks whether at least one element fulfills the search criterion. Entities with certain criteria can also be deleted using `delete` or `remove`.

This information may be followed by any free text, that is, a description. For example, it could start with `findMyPerfectSuperDuperEntityBean`, which would be identical to `find`.

Then comes the separator `By`. This is important because it separates the subject ("What is it about?") from the constraint. The predicate consists of a collection of parts. These parts can be linked to `And`, and these `And`-linked parts can in turn be linked to `Or`.

The predicates access persistent attributes that are specified via parameters. We can also express whether certain persistent attributes are `null` or non-`null`. It's possible to check if strings contain substrings and perform string checks regardless of the case (uppercase/lowercase). After specifying these parts, an `OrderBy` with an ordering criterion can follow, and the method name is completed.

Repository Keywords for Spring Data

The diagram shown earlier in Figure 7.11 shows the keywords for the subject, but doesn't show the keywords and modifiers for the predicates. These are listed in the reference documentation at *https://docs.spring.io/spring-data/commons/docs/current/reference/html/#repository-query-keywords*. Derived query methods are supported in principle by all family members. However, a specific member may not be able to implement all predicate keywords and modifiers. A case-insensitive search could result in a problem (e.g., with a key-value store). In addition, *Spring Data MongoDB* supports `near` and `within` for a geodata query, but other Spring Data family members tend not to do that.

Listing all predicate keywords and modifiers isn't particularly helpful without an example. Therefore, it's useful to study concrete applications in the reference documentation at the Spring Data JPA project at *https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#jpa.query-methods.query-creation* (see Table 7.2).

Keyword	Example	JPQL
`Distinct`	`findDistinctByLastnameAndFirstname`	`select distinct … where x.lastname = ?1 and x.firstname = ?2`
`And`	`findByLastnameAndFirstname`	`… where x.lastname = ?1 and x.firstname = ?2`
`Or`	`findByLastnameOrFirstname`	`… where x.lastname = ?1 or x.firstname = ?2`

Table 7.2 Repository Keywords from the Spring Data JPA Reference Documentation

Keyword	Example	JPQL
`Is`, `Equals`	`findByFirstname`, `findByFirstnameIs`, `findByFirstnameEquals`	… `where x.firstname = ?1`
`Between`	`findByStartDateBetween`	… `where x.startDate between ?1 and ?2`
`LessThan`	`findByAgeLessThan`	… `where x.age < ?1`
`LessThanEqual`	`findByAgeLessThanEqual`	… `where x.age <= ?1`
`GreaterThan`	`findByAgeGreaterThan`	… `where x.age > ?1`
`GreaterThanEqual`	`findByAgeGreaterThanEqual`	… `where x.age >= ?1`
`After`	`findByStartDateAfter`	… `where x.startDate > ?1`
`Before`	`findByStartDateBefore`	… `where x.startDate < ?1`
`IsNull`, `Null`	`findByAge[Is]Null`	… `where x.age is null`
`IsNotNull`, `NotNull`	`findByAge[Is]NotNull`	… `where x.age not null`
`Like`	`findByFirstnameLike`	… `where x.firstname like ?1`
`NotLike`	`findByFirstnameNotLike`	… `where x.firstname not like ?1`
`StartingWith`	`findByFirstnameStartingWith`	… `where x.firstname like ?1 (parameter bound with appended %)`
`EndingWith`	`findByFirstnameEndingWith`	… `where x.firstname like ?1 (parameter bound with prepended %)`
`Containing`	`findByFirstnameContaining`	… `where x.firstname like ?1 (parameter bound wrapped in %)`
`OrderBy`	`findByAgeOrderByLastnameDesc`	… `where x.age = ?1 order by x.lastname desc`
`Not`	`findByLastnameNot`	… `where x.lastname <> ?1`

Table 7.2 Repository Keywords from the Spring Data JPA Reference Documentation (Cont.)

Keyword	Example	JPQL
`In`	`findByAgeIn(Collection<Age> c)`	… `where x.age in ?1`
`NotIn`	`findByAgeNotIn(Collection<Age> c)`	… `where x.age not in ?1`
`True`	`findByActiveTrue()`	… `where x.active = true`
`False`	`findByActiveFalse()`	… `where x.active = false`
`IgnoreCase`	`findByFirstnameIgnoreCase`	… `where UPPER(x. firstname) = UPPER(?1)`

Table 7.2 Repository Keywords from the Spring Data JPA Reference Documentation (Cont.)

Task with Queries from Method Names

We want to add four queries to the `ProfileRepository` to solve different tasks. We're looking for a method that does the following:

- Returns the profile with the longest mane
- Provides all profiles sorted in descending order by mane length
- Determines all profiles that have a mane length greater than a given parameterization
- Delivers a maximum of 10 profiles, which were the last to log in

The first three queries require only the persistent attribute `manelength`, and the last query requires only `lastseen`.

Tip

Those who use the *IntelliJ Ultimate Edition* will be happy about the keyboard completion. The integrated development environment (IDE) displays the persistent attributes and directly checks which queries are possible.

Proposed solution:

```
Optional<Profile> findFirstByOrderByManelengthDesc();
List<Profile>     findByOrderByManelengthDesc();
List<Profile>     findByManelengthGreaterThan( short min );
List<Profile>     findFirst10ByOrderByLastseenDesc();
```

In the first method, `Profiles` would also have been possible as a return type, but if the table is empty, the return would be `null`. `Optional` expresses better that there could be no `Profile`.

Delete with Status

Derived query methods can be handy in different ways. Let's look at the `delete*(…)` methods from the `CrudRepository`:

- `void delete(T entity)`
- `void deleteAll()`
- `void deleteAll(Iterable<? extends T> entities)`
- `void deleteAllById(Iterable<? extends ID> ids)`
- `void deleteById(ID id)`

The drawback of the previously mentioned methods is that they don't provide any information about whether the deletion was successful or not. To solve this problem elegantly, a derived method query can be used:

```
public interface ProfileRepository extends JpaRepository<Profile, Long> {
  int deleteProfileById( long id );
}
```

Actually, we want to override `deleteById(…)`, but changing the return type from `void` to `int` isn't possible. Therefore, the description is added, which creates a new method that returns `int`. The return encodes the number of changed rows and is either `0` or `1`, which tells us whether a row could be deleted or not. All modifying database operations must be transactional.

7.8.3 Returns from Derived Query Methods

The derived query methods can have different return types:

- With a `void`, there is no return if, for example, something is deleted.
- Primitive types and wrapper types are possible, for example, in `count` or in projections.
- Single objects are possible when entities are queried or projection occurs.
- If collections come back, the return type can be `Iterator`, `Collection`, `List`, or `Streamable`.
- To avoid `null` in the return, the `Optional` data type can be used.
- Pagination information can be passed to the derived query methods. Then the result can be of type `Page` or `Slice`.
- A geodata query can also result in `GeoResult<T>`, `GeoResults<T>`, or `GeoPage<T>`.

7.8.4 Asynchronous Query Methods

Derived query methods can be annotated with `@Async`. The results can be received as `Future` or `CompletableFuture`:

```
@Async Future<Profile>            findByNickname( String nickname );
@Async CompletableFuture<Profile> findByNickname( String nickname );
```

Using it requires an `@EnableAsync` that generates proxy objects; we discussed this in Chapter 4, Section 4.3. The queries take place in a background thread and has nothing to do with reactive programming; otherwise, the `Mono` or `Flux` data types would show up.

7.8.5 Streaming Query Methods

A derived query method can return a `Stream`. We can see an example with method `findByOrderByManelength()`:

```
@Transactional( readOnly = true )
Stream<Profile> findByOrderByManelength();
```

The call must take place in a transaction; if you're only reading something, it's sufficient to set the flag `readOnly` to `true`. The call can look like this:

```
try ( Stream<Profile> stream = repository.findByOrderByManelength() ) {
  stream.…
}
```

A `Stream` must be closed at the end in any case. A `try-with-resources` can help.

7.8.6 Advantages and Disadvantages of Derived Query Methods

One of the advantages of derived query methods in Spring Data is that they allow queries to be derived from method names, which is more convenient than manually setting a Native SQL, JPQL, or MongoDB JSON query in the `@Query` annotation. This also avoids any references to specific technologies, making it more in line with *domain-driven design* principles. Additionally, the development environment can assist with writing these queries, making it a major advantage.

However, there are also several disadvantages to derived query methods. Refactoring could potentially result in errors, especially if there are no tests in place to catch them. Additionally, if method names aren't chosen carefully, it could lead to incorrect queries being generated. For example, if the method name is changed from `findProfilesByContainingNickname` to `findByNickname`, it may still be a valid, derived query method, but it no longer searches for substrings. Other renames might be noticeable, such as `findByName(…)`—Spring Boot checks this at startup.

Finally, derived query methods have limitations and can't represent all the features of SQL or JPQL, including nested queries, subqueries, and special joins. Derived query methods should be seen as a nice shortcut for general queries, but nothing more. Because semantic method names are rare in the Java environment, it's also understandable that the people in charge of software architecture don't want to bind a method name to a query.

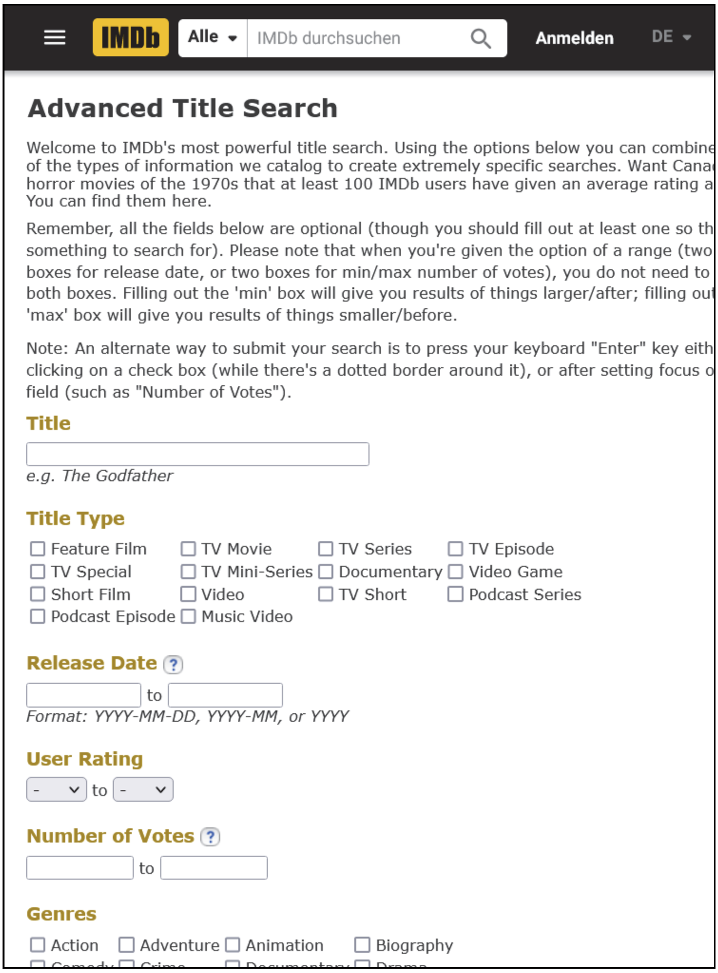
7.9 Criteria API and JpaSpecificationExecutor

In this section, we'll take a look at the *Criteria API* in *Jakarta Persistence* and the Spring-specific `JpaSpecificationExecutor` data type, which simplifies the use of the Criteria API.

The Criteria API solves two problems:

- Queries often have to be composed dynamically, for example, because certain filter criteria are activated.
- Strings aren't type-safe as names for the persistent attributes in JPQL strings. It would be nice to have typed objects where we can't make mistakes in the editor because the compiler can make certain checks.

As an example of the benefits of the Criteria API, we can refer to the search feature on the Internet Movie Database (IMDB). Figure 7.12 displays a screenshot of the search interface on *https://www.imdb.com/search/title/*.



Advanced Title Search

Welcome to IMDb's most powerful title search. Using the options below you can combine of the types of information we catalog to create extremely specific searches. Want Cana horror movies of the 1970s that at least 100 IMDb users have given an average rating a You can find them here.

Remember, all the fields below are optional (though you should fill out at least one so th something to search for). Please note that when you're given the option of a range (two boxes for release date, or two boxes for min/max number of votes), you do not need to both boxes. Filling out the 'min' box will give you results of things larger/after; filling out 'max' box will give you results of things smaller/before.

Note: An alternate way to submit your search is to press your keyboard "Enter" key eith clicking on a check box (while there's a dotted border around it), or after setting focus o field (such as "Number of Votes").

Title

e.g. The Godfather

Title Type

☐ Feature Film ☐ TV Movie ☐ TV Series ☐ TV Episode
☐ TV Special ☐ TV Mini-Series ☐ Documentary ☐ Video Game
☐ Short Film ☐ Video ☐ TV Short ☐ Podcast Series
☐ Podcast Episode ☐ Music Video

Release Date ?

to

Format: YYYY-MM-DD, YYYY-MM, or YYYY

User Rating

- to -

Number of Votes ?

to

Genres

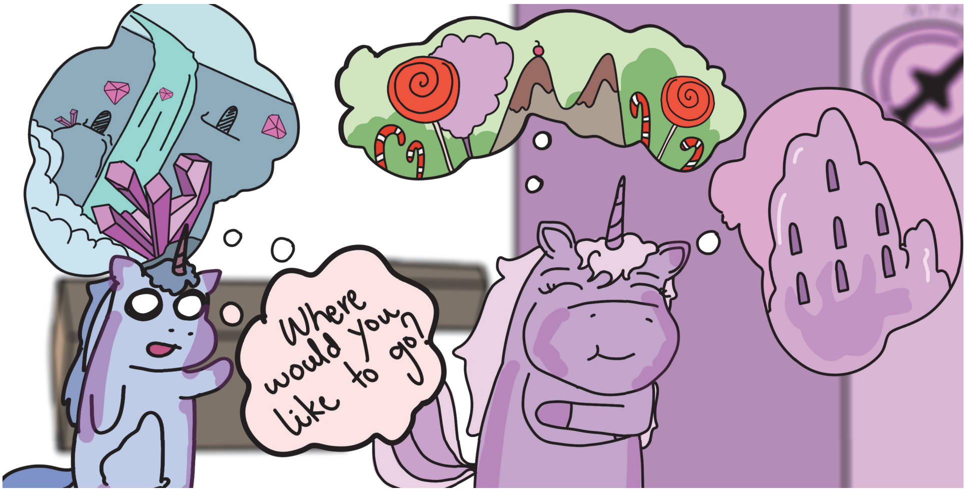
☐ Action ☐ Adventure ☐ Animation ☐ Biography

Figure 7.12 IMDB Web Page for Searching Movies and Series

Through this page, films and series can be found based on certain criteria. If such queries were necessary, we wouldn't have the best tools in hand with our current capabilities, so what could a solution look like?

- We could query all objects via the repository with `findAll()` and then filter them client-side. But this is a bad idea and not an option!
- We could try to work with QBE. The query style would be optimal, but Spring Data is too limited for this in that with a question about the **User Rating**, we don't get further with `QueryByExampleExecutor` because we can only determine one rating, for example, but no ranges of ratings. For instance, we couldn't determine which movies were rated over 8.
- The constraints could be dynamically concatenated in the `WHERE` clause, resulting in a JPQL or SQL string. However, this isn't optimal and is error-prone.

Another option is to prepare an all-inclusive query as a prepared statement and then fill it with three dozen parameters later.


Where would you like to go?

7.9.1 Criteria API

As applications become more complex and have multiple search criteria, the number of methods or parameter lists required to handle them becomes unmanageable, and SQL queries become unwieldy. To address this issue, the Criteria API was introduced in Java Persistence API 2.0. With the Criteria API, search criteria can be written in Java and combined dynamically using logical operators. `EntityManager` receives the query and executes it. However, using the Criteria API through `EntityManager` involves a significant amount of code. To simplify the process, Spring Data JPA provides a streamlined interface for using the Criteria API.

To see the shortcut that Spring Data JPA provides, let's first look at a simple query using the pure Criteria API:

```
Class<Profile> clazz = Profile.class;
///
CriteriaBuilder criteriaBuilder = em.getCriteriaBuilder();
CriteriaQuery<?> criteriaQuery = criteriaBuilder.createQuery( clazz );
Root<?> root = criteriaQuery.from( clazz );
//
Predicate condition = criteriaBuilder.gt( root.get( "manelength" ), 10 );
//
TypedQuery<?> query = em.createQuery( criteriaQuery
                                      // .select( root )
                                      .where( condition ) );
List<?> resultList = query.getResultList();
///
resultList.forEach( profile -> log.info( profile.toString() ) );
```

All profiles that have a mane length greater than `10` are selected. The program contains several blocks, each of which is labeled with comments in the code. Certain expressions can be generated automatically using an `EntityManager` and a `Class` object, including the `CriteriaBuilder`, `CriteriaQuery`, and a `Root` object. While the `Predicate` is specific to each case, execution can be abstracted, which is where the `Specification` data type in Spring Data JPA is useful.

7.9.2 Functional Interface Specification

Functional interface `Specification`[26] specifies an abstract method `toPredicate(…)`, as shown in Figure 7.13.

The custom implementation must realize `toPredicate(…)` and the query. Spring will call the method later and pass an instance of `Root`, `CriteriaQuery`, and `CriteriaBuilder`; Spring will automatically build these objects using Spring's `EntityManager` and `Class` object. After the `Specification` is built, it can be executed.

---

[26] *https://docs.spring.io/spring-data/jpa/docs/current/api/org/springframework/data/jpa/domain/Specification.html*

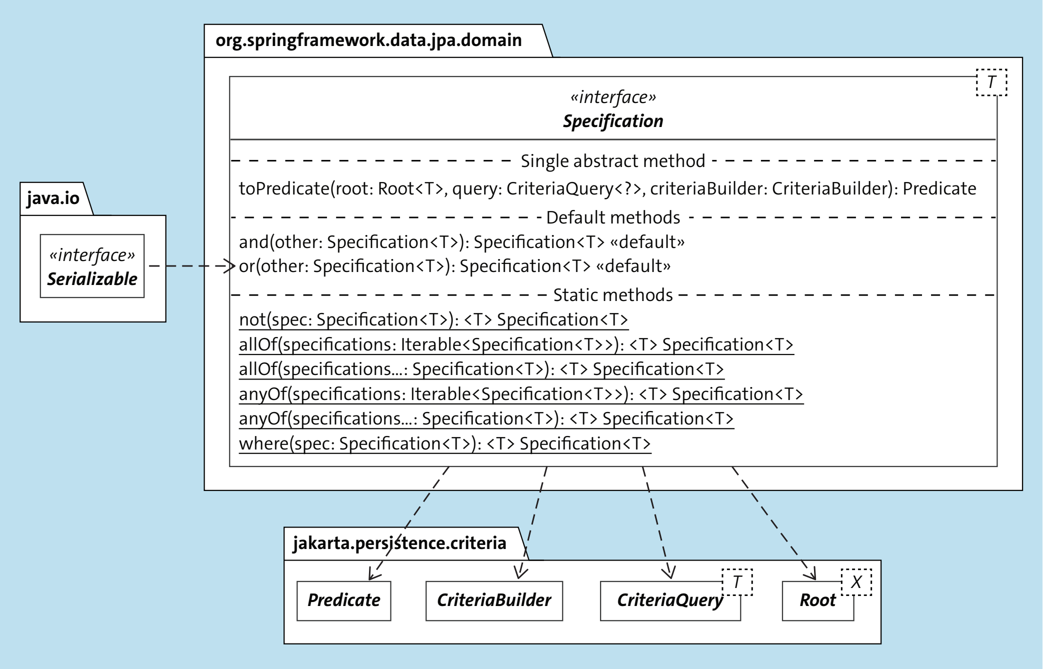

org.springframework.data.jpa.domain
T
«interface»
Specification
Single abstract method
toPredicate(root: Root<T>, query: CriteriaQuery<?>, criteriaBuilder: CriteriaBuilder): Predicate
Default methods
and(other: Specification<T>): Specification<T> «default»
or(other: Specification<T>): Specification<T> «default»
Static methods
not(spec: Specification<T>): <T> Specification<T>
allOf(specifications: Iterable<Specification<T>>): <T> Specification<T>
allOf(specifications...: Specification<T>): <T> Specification<T>
anyOf(specifications: Iterable<Specification<T>>): <T> Specification<T>
anyOf(specifications...: Specification<T>): <T> Specification<T>
where(spec: Specification<T>): <T> Specification<T>
java.io
«interface»
Serializable
jakarta.persistence.criteria
Predicate
CriteriaBuilder
CriteriaQuery
T
Root
X

Figure 7.13 “Specification” Interface

7.9.3 JpaSpecificationExecutor

A `Specification` is executed by a `JpaSpecificationExecutor`.[27] Therefore, our own repository must extend `JpaSpecificationExecutor`. The interface is a bit different from the other Spring Data interfaces, and also `JpaRepository` doesn’t extend `JpaSpecificationExecutor`. Consequently, if we want to use the functionality of `JpaRepository` while also executing a `Specification`, two interfaces need to be extended:

```
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.JpaSpecificationExecutor;

public interface ProfileRepository extends
    JpaRepository<Profile, Long>,
    JpaSpecificationExecutor<Profile> {
}
```

The `JpaSpecificationExecutor` interface doesn’t extend any other repository interface. If we want to use the methods for specifications as well as CRUD operations, we need to inherit from both interfaces. However, if our own repository only wants to use `JpaSpecificationExecutor`, it doesn’t need to extend `JpaRepository` or `[List]CrudRepository`.

---

[27] *https://docs.spring.io/spring-data/jpa/docs/current/api/org/springframework/data/jpa/repository/JpaSpecificationExecutor.html*

Only the generic type information of `Profile` needs to be specified, as `JpaSpecificationExecutor` has nothing to do with IDs.

7.9.4 Methods in JpaSpecificationExecutor

The UML diagram in Figure 7.14 shows the methods of `JpaSpecificationExecutor`.

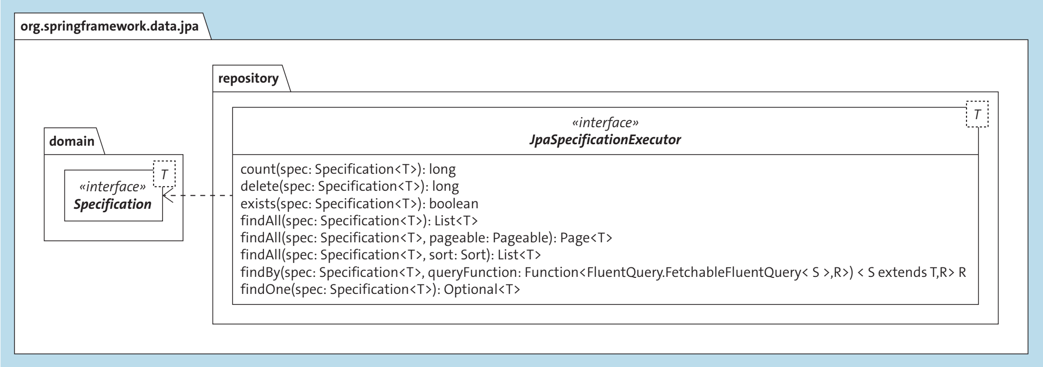

org.springframework.data.jpa
domain
«interface»
Specification
T
repository
«interface»
JpaSpecificationExecutor
T
count(spec: Specification<T>): long
delete(spec: Specification<T>): long
exists(spec: Specification<T>): boolean
findAll(spec: Specification<T>): List<T>
findAll(spec: Specification<T>, pageable: Pageable): Page<T>
findAll(spec: Specification<T>, sort: Sort): List<T>
findBy(spec: Specification<T>, queryFunction: Function<FluentQuery.FetchableFluentQuery< S >,R>) < S extends T,R> R
findOne(spec: Specification<T>): Optional<T>

Figure 7.14 Methods in "JpaSpecificationExecutor"

The methods look familiar if you think of `QueryByExampleExecutor` from Section 7.5. However, the data type there wasn't `Specification`, but `Example`.

7.9.5 Specification Implementations

If our own repository inherits from `JpaSpecificationExecutor`, an implementation of the functional interface `Specification` is required next. The instance is then passed to a method from `JpaSpecificationExecutor`.

Because the `toPredicate(…)` method may also return `null`, the simplest implementation may look like this:

```
Specification<Profile> spec = new Specification<Profile>() {
  public Predicate toPredicate( Root<Profile> root,
                                CriteriaQuery<?> query,
                                CriteriaBuilder builder ) {
    return null;
  }
};
List<Profile> allProfiles = profiles.findAll( spec );
```

If the `Specification` method returns `null`, no filter criterion applies and the result is a list of all profiles.

Because `Specification` is a functional interface, the query can be implemented using a lambda expression:

```
Specification<Profile> spec = ( root, query, builder ) -> null;
List<Profile> allProfiles = profiles.findAll( spec );
```

In the following, the queries will become more realistic and demonstrate the advantage of typing.

Profiles with Long Manes

In the next example, we want to find all the profiles that have a mane length greater than `10`:

```
Specification<Profile> longMane = ( root, query, builder ) ->
  builder.greaterThanOrEqualTo( root.get( "manelength" ), "10" );
List<Profile> longManeProfiles = profiles.findAll( longMane );
```

Each query consists of clauses so that `SELECT`, `FROM`, `WHERE`, `GROUP BY`, `HAVING`, and `ORDER BY` are further restricted via a Java API. Each Java Persistence API query clause is composed of expressions. In our case, the `CriteriaBuilder` helps to build a `WHERE` clause using Java methods. For example, the comparison `a >= b` can be expressed as `greaterThanOrEqualTo(a, b)`. The persistent attribute is accessed through the `Root` object. It doesn't matter whether the length is given as a string or as a numeric value, as in the implementation. If you remove the local variable, the expression is a "one-liner":

```
List<Profile> result = profiles.findAll(
  ( r, __ , b ) -> b.greaterThanOrEqualTo( r.get("manelength"), 10) );
```

While this is a bit of overkill, it points out that Spring Data JPA with the `JpaSpecificationExecutor` makes criteria execution much easier.

The query ultimately generates and processes SQL in the background. While there are still strings in the expression, complete SQL or JPQL is no longer present. If "`manelength`" is written, it remains the name of the persistent attribute and not the column name. However, an execution error will occur if "`maneLength`" is used instead of "`manelength`", for example.

Collection of Specifications

If there are several individual criteria, it's worth creating a class that serves as a collection. This way, you can provide small reusable `Specification` objects:

```
public class ProfileSpecifications {

  private ProfileSpecifications() { }
  public static final Specification<Profile> longMane =
   (root, query, builder) ->
    builder.greaterThanOrEqualTo( root.get("manelength"), "10" );
//    builder.greaterThanOrEqualTo( root.get("manelength"), (short) 10 );

  public static final Specification<Profile> nameContainsFat =
   (root, query, builder) -> {
    Expression<String> nickname  = root.get( "nickname" );
    Expression<String> lowerName = builder.lower( nickname );
    return builder.like( lowerName, "%fat%" );
  };

  public static Specification<Profile> longMane( int length ) {
    return (root, query, builder) ->
      builder.greaterThanOrEqualTo( root.get("manelength"), (short)length );
  }
}
```

The first query is as known; a second `Specification` determines whether the lowercase nickname contains `fat`. The variables are written more explicitly, so you can see the data types well.

Parameterized queries are also conceivable. Therefore, method `longMane(int length)` is added to the two variables. The parameter is used again in `greaterThanOrEqualTo(…)`. The entire thing can be called like this:

```
List<Profile> longManeProfiles =
    profiles.findAll( ProfileSpecifications.longMane( 10 ) );
```

7.9.6 Assemble Specification Instances

Let’s reflect on the web interface used for selecting IMDB, where multiple criteria could be chosen. To achieve this, the chosen elements need to be translated into `Specification` objects that are linked together. Along with the `toPredicate(…)` method, the `Specification` interface includes two built-in methods for linking and one static method for negating:

- `default Specification<T> and(Specification<T> other)`
- `default Specification<T> or(Specification<T> other)`
- `static <T> Specification<T> not(Specification<T> spec)`

The instance methods `and(…)` and `or(…)` combine the current `Specification` with another `Specification` and return a new `specification`.

Consider the following example in which we want to retrieve all profiles that have both a long mane and the word `fat` in their nickname. This can be achieved using the following code:

```
List<Profile> longManeAndNameContainsFat =
    profiles.findAll( longMane.and( nameContainsFat ) );
```

In this code snippet, the `ProfileSpecifications` class and its methods are imported statically. The query is both concise and typed, and it demonstrates how small `Specification` objects can be combined to create larger, more complex ones. This is particularly useful when dealing with filter criteria.

7.9.7 Internals *

The condensation made by Spring Data JPA can be easily read in the code of the `SimpleJpaRepository` class[28] by interested parties. Let's take a look at the `findAll(Specification)` method (with another method in-lined for better readability):

```
public List<T> findAll( @Nullable Specification<T> spec ) {
  return getQuery( spec, getDomainClass(), Sort.unsorted() )
             .getResultList();
}
```

The `getDomainClass()` method returns a `Class` object, which is passed together with the `Sort` object to the `getQuery(…)` method. It returns a `TypedQuery`, and `getResultList()` returns the objects for the return value.

The protected `getQuery(…)` method does what we outlined at the beginning of this section: it builds a `CriteriaBuilder`, `CriteriaQuery`, and `Root` object, and returns the `TypedQuery` using `createQuery(…)`:

```
protected <S extends T> TypedQuery<S> getQuery(
          @Nullable Specification<S> spec, Class<S> domainClass, Sort sort ) {
  CriteriaBuilder builder = em.getCriteriaBuilder();
  CriteriaQuery<S> query = builder.createQuery( domainClass );
  Root<S> root = applySpecificationToCriteria( spec, domainClass, query );
  query.select( root );
  if ( sort.isSorted() )
    query.orderBy( toOrders( sort, root, builder ) );

  return applyRepositoryMethodMetadata( em.createQuery( query ) );
}
```

---

[28] *https://github.com/spring-projects/spring-data-jpa/blob/main/spring-data-jpa/src/main/java/org/springframework/data/jpa/repository/support/SimpleJpaRepository.java*

The private method `applySpecificationToCriteria(…)` builds the `Root` object and applies our `Specification`:

```
private <S, U extends T> Root<U> applySpecificationToCriteria( @Nullable Specifi
cation<U> spec, Class<U> domainClass, CriteriaQuery<S> query ) {
  Root<U> root = query.from( domainClass );
  if ( spec == null )
    return root;
  CriteriaBuilder builder = em.getCriteriaBuilder();
  Predicate predicate = spec.toPredicate( root, query, builder );
  if ( predicate != null )
    query.where( predicate );
  return root;
}
```

Here, we find exactly the callback to `toPredicate(…)` into our code. The implementation also shows the following: if the `Predicate` is `null`, there is no `query.where(predicate)`.

7.9.8 Metamodel Classes

While the Criteria API offers type safety in several areas, expressions such as `root.get("manelength")` and `root.get("nickname")` can cause problems. If the nickname field is renamed to name, for example, any errors will likely only be detected during testing or, in the worst-case scenario, in production.

To solve this problem, a *metamodel* can be used. The term “meta” refers to “information about information” or “data about data.” In the case of entity beans, the metamodel consists of special abstract classes that contain a variable for each persistent attribute. These variables are then accessed in a typed manner later on. This provides improved type safety and reduces the risk of errors caused by changes in the codebase.

Maven Plugin Hibernate jpamodelgen

In principle, you could write the metamodel classes manually, but this would still be error-prone, so you generate the classes. Each Jakarta Persistence provider comes with a generator; for Hibernate, it’s *Hibernate Jpamodelgen* described at *https://docs.jboss.org/hibernate/orm/current/userguide/html_single/Hibernate_User_Guide.html#tooling-modelgen*.

To integrate the model generator into the build process, you can either use a direct dependency or an annotation processor—the preferred approach when dealing with multiple annotation processors. For Maven, you add the following configuration to the POM file for the Maven Compiler Plugin:[29]

---

[29] Formerly under `org.hibernate:hibernate-jpamodelgen`, so don’t get that confused.

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <configuration>
        <annotationProcessorPaths>
          <! -- ... -->
          <path>
            <groupId>jakarta.persistence</groupId>
            <artifactId>jakarta.persistence-api</artifactId>
            <version>${jakarta-persistence.version}</version>
          </path>
          <path>
            <groupId>org.hibernate.orm</groupId>
            <artifactId>hibernate-jpamodelgen</artifactId>
            <version>${hibernate.version}</version>
          </path>
        </annotationProcessorPaths>
      </configuration>
    </plugin>
    <!-- spring-boot-maven-plugin -->
  </plugins>
</build>
```

That's all there is to it! When the next build is performed, Maven will generate the metaclasses in the *target* directory under *classes*. For each entity bean, a class with the metamodel is created. These classes have the same name as the entity bean classes, but end with an underscore. The metamodel generator will place source code in the *generated_sources/annotations* subdirectory during the build. Let's examine a portion of the *Profile_.java* file:

```
@StaticMetamodel(Profile.class)
public abstract class Profile_ {
  public static volatile SingularAttribute<Profile, LocalDateTime> lastseen;
  public static volatile SingularAttribute<Profile, LocalDate> birthdate;
  public static volatile SingularAttribute<Profile, Byte> gender;
  public static volatile SingularAttribute<Profile, String> nickname;
  public static volatile SingularAttribute<Profile, Byte> attractedToGender;
  public static volatile SingularAttribute<Profile, String> description;
  public static volatile SetAttribute<Profile, Profile> profilesThatLikeMe;
  public static volatile SingularAttribute<Profile, Short> manelength;
  public static volatile SetAttribute<Profile, Profile> profilesThatILike;
  public static volatile SingularAttribute<Profile, Long> id;
  public static volatile SingularAttribute<Profile, Unicorn> unicorn;
  public static volatile ListAttribute<Profile, Photo> photos;
  public static final String LASTSEEN = "lastseen";
  public static final String BIRTHDATE = "birthdate";
  public static final String GENDER = "gender";
  public static final String NICKNAME = "nickname";
  public static final String ATTRACTED_TO_GENDER = "attractedToGender";
  public static final String DESCRIPTION = "description";
  public static final String PROFILES_THAT_LIKE_ME = "profilesThatLikeMe";
  public static final String MANELENGTH = "manelength";
  public static final String PROFILES_THAT_ILIKE = "profilesThatILike";
  public static final String ID = "id";
  public static final String UNICORN = "unicorn";
  public static final String PHOTOS = "photos";
}
```

In the metamodel generated by Maven, there is a variable for each persistent attribute in the corresponding entity bean. For simple attributes that aren't collections, type `SingularAttribute` is used. However, for associations that are of type `Collection`, `CollectionAttribute` is used. For a `Set`, the type is `SetAttribute`; for a `List`, it's `ListAttribute`; and for a `Map`, it's `MapAttribute`.

Additionally, string constants are initialized with the corresponding name of the persistent attribute. These constants can be used instead of hardcoding string literals in Criteria API queries to reference the persistent attributes, providing improved type safety and reducing the risk of errors due to changes in the codebase.

get(SingularAttribute) from Path (Base Type of Root)

Instead of using the `root.get(String)` method, the generated `*attribute` variables can be passed in `get(…)`. For example, instead of `root.get("manelength")`, we can use `root.get(Profile_.manelength)`.

There are several advantages to using the static variables generated by the metamodel. First, it eliminates the possibility of typos and mistakes, and keyboard completion can be used to quickly and accurately reference the persistent attributes. However, the more significant advantage is that the metamodel is automatically generated, which means that if the entity bean changes (e.g., because of renaming), the metamodel classes will also be updated accordingly. In contrast, if there is a reference to a persistent attribute in your program, and the attribute is later renamed or removed, the code will continue to compile without errors until runtime, where it may cause problems or unexpected behavior.

Using a metamodel ensures that any changes to the entity bean are immediately reflected in the Criteria API queries, and any invalid references to renamed or removed attributes are caught by the compiler, resulting in red squiggles in the code that indicate the need for fixes. In conclusion, it's highly recommended for anyone working with the Criteria API to use a metamodel for improved type safety and reduced risk of errors due to changes in the codebase.

7.9.9 Cons of Using the Criteria API

Although the Criteria API offers the benefit of designing queries dynamically and detecting changes during refactoring, its performance may pose some concerns. The dynamic query needs to be translated into SQL, which may not always result in optimal prepared statements.

The whole thing depends a bit on the provider, but Hibernate 6 does the following for each request:

1. An object tree is constructed from the linked criteria. The internal data structure is called Hibernate *Semantic Query Model* (SQM).
2. Hibernate translates the SQM into SQL.[30]
3. The database executes the SQL.

Hibernate has another feature that can also be a performance bottleneck. By default, Hibernate doesn't use bind variables to convert numeric values to prepared statements, but instead directly inserts the values into the SQL string. This can be a performance bottleneck because it floods the database with new queries and can cause the prepared statement cache to overflow. For instance, if a query involves mane length, which is a numeric value, the generated SQL would contain the actual value, leading to this issue.

This property can be changed in Spring Boot via a configuration property, for example, in the familiar *application.properties* file:

```
spring.jpa.properties.hibernate.criteria.literal_handling_mode=bind
```

Possible values of `LiteralHandlingMode` are `AUTO` (strings via bind variables, numeric values not), `INLINE` (no bind variables, everything in SQL strings), `BIND` (everything via bind variables). Performance tests must show with which settings the application is fastest.

To sum up, although the Criteria API offers dynamic query design and easy refactoring, its performance may not be optimal due to SQL translation and nonoptimal prepared statements. Therefore, it's crucial to conduct tests to determine if the Criteria API is the best option for the current project. For achieving maximum performance, custom prepared statements could be a better alternative.

---

[30] Before version 6, Hibernate first generated a JPQL string from the `Criteria object tree`, which was decomposed again in the next step and then converted into SQL. This detour via the string is omitted with the SQM.

The Criteria API is part of Jakarta Persistence, but not the only solution to realize typesafe queries. We'll get to know another API outside the standard with Querydsl.

7.10 Alternatives to JDBC Jakarta Persistence

So far, we've covered most of the Spring-specific methods for connecting to relational databases. We began with the `JdbcTemplate`, then progressed to the `EntityManager`, and from there to the Spring Data JPA project, which introduced us to repositories. We explored various repository types for queries and used the Criteria API to perform typed queries with minimal JPQL and more Java code. These methods adhere to industry standards.

In addition to the Jakarta EE standard and Spring's offerings, there are other libraries available. Over the years, several alternatives have emerged that aim to move away from error-prone SQL strings and enable typed queries. These libraries fall between the raw JDBC API and the more advanced Jakarta Persistence, as shown in Figure 7.15.

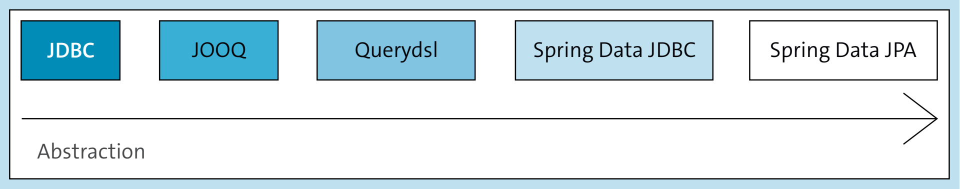


Figure 7.15 Solutions between JDBC and Jakarta Persistence

Ordered from low to high level of abstraction, JDBC is on the left and Jakarta Persistence is on the right, of which, the Spring project supports Spring Data JPA and already upgrades with the repositories.

In addition to the options discussed so far for connecting to relational databases with Spring, there are other libraries available. One such library is *jOOQ* (*www.jooq.org*), which focuses on making SQL queries using a special domain-specific language (DSL) in Java. In jOOQ, each SQL keyword is represented as a Java method, and a metamodel is used so that column names don't have to be hard-coded as strings in the source code.

Another popular alternative is *Querydsl*, which also uses a metamodel and allows for changes to be made dynamically. Although "Query" is in the project name, Querydsl isn't limited to querying and can also be used for updates and deletes.

Spring Data JDBC is another member of the Spring Data project family, which provides a good solution when features such as lazy loading or dirty checking aren't needed, but the Spring Data repository abstraction is desired. Querydsl sits in the middle of the figure, but it's actually integrated with Spring Data JPA, providing access to Querydsl for Java Persistence API–based repositories.

7.10.1 Querydsl

The goal of the open-source library Querydsl (*http://querydsl.com*) is to realize typesafe queries via a generated metamodel. These queries are realized in Java, whereby we fall back here on a Fluent API so that it looks also almost as if we are writing SQL directly in the Java code.

Here's an example of the Querydsl API:

```
var query = new JPAQueryFactory( em );
QProfile profile = QProfile.profile;
List<Profile> persons =
    query.selectFrom( profile )
         .where(
            profile.manelength.gt( 10 ),
            profile.birthdate.after( LocalDate.now().minusYears(60) ) )
         .orderBy( profile.manelength.asc(),
                  profile.lastseen.desc() )
         .fetch();
```

In the Querydsl API, the first step is to create a query object, which in the case of Jakarta `Persistence` is the `JPAQueryFactory`. This object is initialized with the `EntityManager` and provides access to the database. Different factory classes exist to support other databases or APIs such as MongoDB.

Thereafter, the program references the static property of the metamodel class `QProfile`, which is automatically generated by the Maven plugin. The generated metamodel is used to avoid using strings for column names, which reduces the risk of typos and errors. The query object is then used to construct the query, which resembles SQL with the `selectFrom(…)` method. Constraints are applied in the `where(…)` method.

Finally, the result is sorted using the `orderBy(…)` method. The `fetch()` method is called to obtain the data as a list. Querydsl creates an internal query metadata object from the information and generates, for example, JPQL. The generated JPQL goes as SQL to the database. Querydsl also provides `*Serializer` classes to create SQL statements for non-Java Persistence API use cases.

Querydsl Metamodel

The metamodel plays a crucial role in Querydsl and is more complex than the Criteria API's metamodel. In the Criteria API, the metamodel classes are simple, containing only attribute variables for types and string constants for the names of the persistent attributes. These `*Attribute` types don't offer any valuable methods. However, in Querydsl, the metamodel includes `*Path` objects that have significantly more functionality. For instance, for a numeric persistent attribute, the `NumberPath` type has around 100 methods that allow comparisons, such as greater than, less than, greater than or equal to, and so on. This is the main difference between Querydsl and the Criteria API. In

Querydsl, the methods are distributed in a more object-oriented way among the `*Path` objects, while in the Criteria API, all methods are located at the `QueryBuilder`. For example, in Querydsl, we saw `profile.manelength.gt(10)`, while in the Criteria API, it would be expressed as `where(criteriaBuilder.gt(root.get("manelength"), 10))`.

Querydsl in the Project Object Model: Dependencies

To use and activate Querydsl, a dependency on the corresponding data types is first necessary. This also includes the implementation that ultimately generates JPQL from the queries:

```
<dependency>
  <groupId>com.querydsl</groupId>
  <artifactId>querydsl-jpa</artifactId>
  <version>${querydsl.version}</version>
  <classifier>jakarta</classifier>
</dependency>
```

Listing 7.3 pom.xml Extension

In addition, an *annotation processor* must be included to generate the metamodel: In `<build><plugins>` we configure the Java Compiler:

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-compiler-plugin</artifactId>
  <configuration>
    <annotationProcessorPaths>
      <!-- … -->
      <path>
        <groupId>jakarta.persistence</groupId>
        <artifactId>jakarta.persistence-api</artifactId>
        <version>${jakarta-persistence.version}</version>
      </path>
      <path>
        <groupId>com.querydsl</groupId>
        <artifactId>querydsl-apt</artifactId>
        <version>${querydsl.version}</version>
        <classifier>jakarta</classifier>
      </path>
    </annotationProcessorPaths>
  </configuration>
</plugin>
```

Listing 7.4 pom.xml Extension

The generator creates *Q-types* (Querydsl-types), which are similar in function to the underscore types of the Criteria API. By default, the metamodel classes come in the *target/generated-sources/annotations* directory. They are in the same package as the entity beans.

The generated class `QUnicorn` looks like this:

```
/** * QUnicorn is a Querydsl query type for Unicorn */
@Generated("com.querydsl.codegen.EntitySerializer")
public class QUnicorn extends EntityPathBase<Unicorn> {
 private static final long serialVersionUID = 1495282826L;
 public static final QUnicorn unicorn = new QUnicorn("unicorn");

 public final NumberPath<Short> attractedToGender = ↩
   createNumber("attractedToGender", Short.class);
 public final DatePath<java.time.LocalDate> birthdate = ↩
   createDate("birthdate", java.time.LocalDate.class);
 public final StringPath description = createString("description");
 public final StringPath email = createString("email");
 public final EnumPath<Gender> gender = createEnum("gender", Gender.class);
 public final NumberPath<Short> manelength = ↩
   createNumber("manelength", Short.class);
 public final NumberPath<Long> id = createNumber("id", Long.class);
 public final DateTimePath<java.time.LocalDateTime> lastseen = ↩
   createDateTime("lastseen", java.time.LocalDateTime.class);
 public final StringPath nickname = createString("nickname");
 public final StringPath password = createString("password");
 public final ListPath<Photo, QPhoto> photos =
 this.<Photo, QPhoto>createList("photos", Photo.class, QPhoto.class, ↩
   PathInits.DIRECT2);

 public QUnicorn(String variable) {
  super(Unicorn.class, forVariable(variable));
 }
 public QUnicorn(Path<? extends Unicorn> path) {
  super(path.getType(), path.getMetadata());
 }
 public QUnicorn(PathMetadata metadata) {
  super(Unicorn.class, metadata);
 }
}
```

Clearly, the `*Path` objects are visible for all persistent attributes.

Spring Data JPA and Querydsl

In principle, we can have Spring inject us with an `EntityManager` and use it to build the `JPAQueryFactory` object and take the code from the beginning of the section. But this is precisely what isn't necessary with *Spring Data JPA* because Querydsl is supported out of the box. There are three special packages:

- `org.springframework.data.querydsl`
- `org.springframework.data.querydsl.binding`
- `org.springframework.data.web.querydsl`

We'll take a closer look at three data types:

- **`QuerydslPredicateExecutor`** is closely related to `JpaSpecificationExecutor`, except that the `QuerydslPredicateExecutor` data type passes `Predicate` objects instead of `Specification` objects. The special repository returns results that match the predicates.
- **`QuerydslRepositorySupport`** is a base class for custom repository implementations. This is useful if you want to implement modifications with Querydsl as well because the `QuerydslPredicateExecutor` can only read entities, but not modify, create, or delete them.
- **`QSort`** can be used to build type-safe sort criteria. And because `QSort` is a subtype of `Sort`, the Querydsl type can be used where a `Sort` object is also possible, such as with a `Pageable` or directly with the `PagingAndSortingRepository`. Using Q-types without strings looks like this:

  ```
  Sort sort = QSort.by( QUnicorn.nickname.asc() )
                   .and( QSort.by( QUnicorn.manelength.desc() ) );
  ```

QuerydslPredicateExecutor

The methods of `QuerydslPredicateExecutor`[31] are similar to the methods in `JpaSpecificationExecutor`, only everything runs through a `Predicate`:

```
package org.springframework.data.querydsl;

import …

public interface QuerydslPredicateExecutor<T> {

  long count(Predicate predicate);
  boolean exists(Predicate predicate);

  Iterable<T> findAll(Predicate predicate);
```

---

[31] *https://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/querydsl/QuerydslPredicateExecutor.html*

```
  Iterable<T> findAll(Predicate predicate, Sort sort);
  Iterable<T> findAll(Predicate predicate, OrderSpecifier<?>... orders);
  Iterable<T> findAll(OrderSpecifier<?>... orders);
  Page<T>     findAll(Predicate predicate, Pageable pageable);

  R findBy(Predicate predicate,
           Function<FluentQuery.FetchableFluentQuery<S>,R> queryFunction);
  Optional<T> findOne(Predicate predicate);
}
```

Data type `Predicate`[32] comes from Querydsl and therefore not from `java.util.function`!

If the `QuerydslPredicateExecutor` is to be used, a repository will need to extend this interface. This may look like this:

```
public interface ProfileRepository
    extends JpaRepository<Profile, Long>,
            QuerydslPredicateExecutor<Profile> {
}
```

The `JpaRepository` and `QuerydslPredicateExecutor` types don't have to be used together. Our application can also use only `QuerydslPredicateExecutor` if that is desired.

The extended `ProfileRepository` can be injected into a client, and then the `QuerydslPredicateExecutor` methods are available:

```
@Autowired ProfileRepository profiles;
```

Suppose we want to find all profiles with a mane length greater than or equal to 10. Then, we would write:

```
Iterable<Profile> longMane =
    profiles.findAll( QProfile.profile.manelength.goe( 10 ) );
longMane.forEach( System.out::println );
```

Task: Use QuerydslPredicateExecutor for a New Shell Query

Let's write a new shell command `profiles-between <min> <max>` that determines the profiles with a mane length between `<min>` and `<max>`.

Proposed solution:

```
@ShellMethod(
 "Display profiles with mane length between a given min and max value"
)
public void profilesBetween( short min, short max ) {
```

---

[32] *https://querydsl.com/static/querydsl/latest/apidocs/com/querydsl/core/types/Predicate.html*

```
  profiles.findAll( QProfile.profile.manelength.between( min, max ) );
          .forEach( System.out::println );
}
```

Querydsl generates JPQL at runtime based on the query, which in turn leads to SQL. Unlike some other query-building methods, strings aren't used in the query. This provides some level of type safety. For instance, if the type of the `manelength` attribute changes, some methods may be missing, and this will become evident through a compiler error.

BooleanBuilder

So far, we've written a single query, but queries can be linked:

```
QProfile profile = QProfile.profile;
var result =
  profiles.findAll( profile.manelength.gt( 1 )
                                   .and( profile.attractedToGender.isNull() );
```

It's mure readable if you extract the predicates. Then, they can be put into a helper class:

```
BooleanExpression profileHasManelengthGt1 = profile.manelength.gt( 1 );
BooleanExpression profileIsBi = profile.attractedToGender.isNull();
```

The variables here are of type `BooleanExpression`,[33] and the class implements `Predicate`.

There is a second possibility, which is helpful when criteria are dynamically linked. Recall the IMDB search from Section 7.9, where different filter criteria can be selected. For Querydsl, if the individual criteria are `BooleanExpressions`, they can be linked with `BooleanBuilder`,[34] which looks like this in pseudocode:

```
var booleanBuilder = new BooleanBuilder();
if ( criterion desired )
  booleanBuilder.and( profileHasManelengthGt1 );
if ( criterion desired )
  booleanBuilder.and( profileIsBi );
var result = profiles.findAll( booleanBuilder );
```

The `BooleanBuilder` is itself a `Predicate`, so it can be passed in all `QuerydslPredicateExecutor` methods.

---

[33] *http://querydsl.com/static/querydsl/latest/apidocs/com/querydsl/core/types/dsl/BooleanExpression.html*

[34] *http://querydsl.com/static/querydsl/latest/apidocs/com/querydsl/core/BooleanBuilder.html*

Other Querydsl Possibilities

Querydsl is an exciting project this is actually more than 20 years old. The reference documentation at *https://querydsl.com/static/querydsl/5.0.0/reference/html_single/* provides information about other possibilities. Some of them will be briefly touched upon here:

- In our queries, we get entity beans, but as in JPQL, projections can provide arbitrary answers. Querydsl has tuple objects. They are then called `QTuple` to match the naming convention.
- For a projection, special parameterized constructors can be called, similar to the constructor expressions in JPQL. The parameterized constructor for projection is annotated with `@QueryProjection`.
- Another interesting feature is delegate methods. The annotation processing tool (APT) generator builds Q-type classes in the source code, and we can add our own methods in the generated classes. This is useful, for example, to support custom data types. For example, if a number contains a monetary amount, we could write conversion methods to another currency.
- For connecting Spring Data repositories and Querydsl, there is an interesting open-source project: *Infobip Spring Data Querydsl* (*https://github.com/infobip/infobip-spring-data-querydsl*). This is definitely worth a look for those who want even more functionality and want to use Querydsl with Spring Data. Another advantage of the project is the support of modern *Reactive Relational Database Connectivity* (R2DBC), which means that reactive queries with Querydsl are possible.

Evaluation of Querydsl

Querydsl offers an intuitive and user-friendly syntax that is easy to read and write. Its comprehensive metamodel enables building highly maintainable and extensible type-safe queries. As it's integrated into the Spring Data JPA, no external integrations are required, making it a highly convenient option for developers. While it supports various data sources, the further development for these sources isn't as clear as it is for Jakarta Persistence API.

One of the major disadvantages of Querydsl is that it's not an official standard such as the Criteria API, but this is usually not a problem for open-source libraries with significant community support. However, Querydsl is highly modularized, and it's unclear which module is maintained and how well. In principle, Querydsl supports various data sources. Besides the Jakarta Persistence API, SQL and MongoDB are among them. This raises questions about its reliability and future development. The absence of any updates since July 2021 (as of April 2023) has led to significant concern about the project's progress, with developers unsure of the extent to which it's still being maintained. While the number of issues is being gradually reduced, the standstill in updates doesn't inspire confidence in the project's longevity. Fortunately, the number of issues is constantly being reduced, so something is happening behind the scenes.[35]

Despite these issues, Querydsl remains a popular and well-tested solution for Jakarta Persistence. Those who are comfortable with SQL will find Querydsl's Fluent API an easy transition. Its code is more concise and easier to read than comparable code using the Criteria API, making it a suitable option for developers who can find alternatives for bugs and are comfortable with its current level of development.

7.10.2 Spring Data JDBC

The Spring Data ecosystem includes various members, and one of them is Spring Data JDBC (*https://spring.io/projects/spring-data-jdbc*), which provides a lightweight alternative to Spring Data JPA. Unlike Spring Data JPA, Spring Data JDBC focuses on SQL as a center of persistence rather than an abstraction such as JPQL. It doesn't define a persistence context, doesn't have mechanisms for checking changed states, and doesn't implement lazy loading for relationships. While this may seem disadvantageous, it actually reduces errors. Unlike `EntityManager`, which can "run full" with managed entity beans, this can't happen with Spring Data JDBC. Moreover, anyone who can write JPQL expressions can also write SQL expressions.

Aggregates

Spring Data JDBC focuses on loading *aggregates*, which is also an important principle that should have been the focus of Spring Data JPA. Aggregates are composites of related objects, where the entry point is the *aggregate root*. The repository usually takes care of the aggregate root, and this aggregate is always processed as a whole. For example, for a 1:1 relationship, the two related entities are part of the same aggregate. Similarly, for a 1:n relationship, all the children are also part of the aggregate and are existence-dependent. However, this means that many-to-one and many-to-many relationships can't be implemented using Java references because it would create a reference to the aggregate from "outside," which isn't allowed. Instead, the native key types such as long, string, or UUID are used on the Java side. It's recommended to avoid bidirectional relationships because a reference from the bottom-up points back to the aggregate root, but this is a bit of a matter of taste.

Spring Data JDBC Dependency

Suppose we intend to migrate a project from Spring Data JPA to Spring Data JDBC. The first thing we would need to do is change the dependency from *Spring Boot Starter Data JPA* to *Spring Boot Starter Data JDBC*:

---

[35] At *https://github.com/querydsl/querydsl/discussions/3368#discussioncomment-3915592*, one of the lead developers writes: "The few bugs that have been reported since have been very minor, non-blocking to the vast majority of users."

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-jdbc</artifactId>
</dependency>
```

Changes to the Entity Class

Spring Data JDBC also uses entities, but it uses different annotations than Jakarta Persistence. Switching is easy because the annotations are often called the same, but come from other packages (e.g., `@Id`, `@Column`, and `@Table`). Because much is also implicit in Spring Data JDBC, many specifications—just like in Jakarta Persistence—aren't explicitly necessary.

Here's an example:

```
import org.springframework.data.annotation.Id;
import org.springframework.data.relational.core.mapping.Table;

// @Table("profile")
// @AccessType( Type.FIELD )   // FIELD is the default
public class Profile {
  @Id
  // @Column("id")
  Long id;

  // @Transient for non-persistent attribute
  …
}
```

You could use `@Table`[36] to determine the table name, but if the class is already named `Profile`, then Spring Data JDBC assumes the table is named `Profile` as well. The same is true for columns: If the column is named the same as the persistent attribute, then the `@Column` annotation can be omitted. If the column is named differently, `@Column`[37] can explicitly assign that. By the way, the `@Id` annotation doesn't come from Spring Data JDBC, but from the Spring Data Commons project.

By default, Spring Data JDBC takes the data from the instance variables—this doesn't need to be set. If certain information should not be persisted, annotation `@Transient` will mark these instance variables.

Besides the annotations, minor adjustments are needed for the attributes. Spring Data JDBC supports enumeration types, but it only maps to the name of the constant; `@Enumerated(EnumType.ORDINAL)` doesn't exist. There are also no annotations such as

---

[36] *https://docs.spring.io/spring-data/jdbc/docs/current/api/org/springframework/data/relational/core/mapping/Table.html*

[37] *https://docs.spring.io/spring-data/jdbc/docs/current/api/org/springframework/data/relational/core/mapping/Column.html*

`@OneToOne`, `@OneToMany`, `@ManyToOne`, and `@ManyToMany`. Of course, associations can be built in principle, but due to the philosophy of aggregates, only 1:1 and 1:n relationships are used; for the latter, Java data types `Set`, `List`, and `Map` are supported. These association types are also evaluated automatically without annotations. In this case, however, the columns must be specially named. The column names aren't always as Spring Data JDBC would like them to be. But via `@MappedCollection`,[38] the column names can be set explicitly. Here's an example:

```
class Profile {
  …
  @MappedCollection( idColumn = "profile_fk", keyColumn = "id" )
  List<Photo> photos;
  …
}
```

There are attribute converters in Jakarta Persistence that prepare specific Java data types into JDBC data types for the Jakarta Persistence provider. These converters don't exist with Spring Data JDBC, but something comparable (i.e., read-write converters) and converters from the `ConversionService` can also be included. These are so far the changes that need to be made to the entity.

Spring Data JDBC also has its own API, of course. There are two options to choose from, which we'll look at now.

JdbcAggregateTemplate and Query

The first option is provided by `JdbcAggregateTemplate`.[39] The object is provided as a Spring-managed bean directly, and we could have it injected:

```
@Autowired JdbcAggregateTemplate jdbcTemplate;
```

The methods are self-explanatory and reminiscent of `CrudRepository`'s methods:

- `long count(Class<?> domainType)`
- `<T> long count(Query query, Class<T> entityClass)`
- `<S> void delete(S aggregateRoot, Class<S> domainType)`
- `void deleteAll(Class<?> domainType)`
- `<T> void deleteAll(Iterable<? extends T> instances, Class<T> domainType)`
- `<T> void deleteAllById(Iterable<?> ids, Class<T> domainType)`
- `long count(Class<?> domainType)`
- `<T> long count(Query query, Class<T> domainType)`

---

[38] *https://docs.spring.io/spring-data/jdbc/docs/current/api/org/springframework/data/relational/core/mapping/MappedCollection.html*

[39] *https://docs.spring.io/spring-data/jdbc/docs/current/api/org/springframework/data/jdbc/core/JdbcAggregateTemplate.html*

- `<S> void delete(S aggregateRoot)`
- `void deleteAll(Class<?> domainType)`
- `<T> void deleteAll(Iterable<? extends T> instances)`
- `<T> void deleteAllById(Iterable<?> ids, Class<T> domainType)`
- `<S> void deleteById(Object id, Class<S> domainType)`
- `<T> boolean exists(Query query, Class<T> domainType)`
- `<T> boolean existsById(Object id, Class<T> domainType)`
- `<T> Iterable<T> findAll(Class<T> domainType)`
- `<T> Page<T> findAll(Class<T> domainType, Pageable pageable)`
- `<T> Iterable<T> findAll(Class<T> domainType, Sort sort)`
- `<T> Iterable<T> findAll(Query query, Class<T> domainType)`
- `<T> Page<T> findAll(Query query, Class<T> domainType, Pageable pageable)`
- `<T> Iterable<T> findAllById(Iterable<?> ids, Class<T> domainType)`
- `<T> T findById(Object id, Class<T> domainType)`
- `<T> Optional<T> findOne(Query query, Class<T> domainType)`
- `<T> T insert(T instance)`
- `<T> T save(T instance)`
- `<T> Iterable<T> saveAll(Iterable<T> instances)`
- `void setEntityCallbacks(EntityCallbacks entityCallbacks)`
- `void setEntityLifecycleEventsEnabled(boolean enabled)`
- `<T> T update(T instance)`

These methods have a special feature: the domain type must always be passed, that is, the type of the entity bean.

The methods to which a `Query`[40] can be passed are intereseting. Queries are built using a Fluent API and are roughly reminiscent of Querydsl and the Criteria API. `Query` is a class, but there are no constructors. Instead, instances are built with two factory methods:

- `static Query query(CriteriaDefinition criteria)`
- `static query empty()`

In the static `query(…)` method, a `CriteriaDefinition`[41] is specified which ultimately leads to an SQL `WHERE` clause. The `CriteriaDefinition` interface is implemented by the class `Criteria`. These objects, in turn, can be constructed using a `from(…)` method. The Javadoc contains an example:

---

[40] *https://docs.spring.io/spring-data/jdbc/docs/current/api/org/springframework/data/relational/core/query/Query.html*

[41] *https://docs.spring.io/spring-data/jdbc/docs/current/api/org/springframework/data/relational/core/query/CriteriaDefinition.html*

```
Criteria.from(Criteria.where("name").is("Foo"),
  Criteria.from(Criteria.where("age").greaterThan(42)) );
```

If you import the methods statically, the code becomes short. With the Fluent API, the criteria can be dynamically assembled again with AND and OR. When we've built the `Query` object, we can send it, for example, with the methods `count(…)`, `exists(…)`, `findAll(…)`, or `findOne(…)`.

Spring Data JDBC Repository

Spring Data JDBC offers a programmatic approach with `JdbcAggregateTemplate`, but it also provides declarative repositories. However, the repositories in Spring Data JDBC differ from those in other Spring Data modules, as follows:

- The first change is to the base interfaces because Spring Data JDBC doesn't currently bring its own `*Repository` interface, so we inherit from `[List]CrudRepository` or `QueryByExampleExecutor`. (The `SimpleJdbcRepository` class actually implements all the methods then, comparable to `SimpleJpaRepository`.)
- In the SQL of `@Query`, only named parameters can be used—no index access—because Spring Data JDBC works internally with `NamedParameterJdbcTemplate`. This is certainly not a limitation, but improves readability.
- As discussed earlier, the derived query method is a special feature of Spring Data. Currently, there is the restriction that only `SELECT` statements can be built, but no modifying or deleting operations.
- There are named queries, but the mapping between names and SQL isn't set at the entity bean via an annotation; instead, it's placed in a *META-INF/jdbc-named-queries.properties* file.[42]
- Because there is no Criteria API, you have to implement dynamic queries with `JdbcAggregatTemplate`.

The reference documentation *https://docs.spring.io/spring-data/jdbc/docs/current/reference/html/* shows what other possibilities Spring Data JDBC offers. Overall, Spring Data JDBC is quite an exciting project, and the reference documentation is also a quick read.

Tip: Switch on Logging

Because Spring Data JDBC is an object-relational mapper (O/R mapper), SQL is generated automatically. If you want to see the generated SQL, set the following configuration properties:

```
logging.level.org.springframework.jdbc.core.JdbcTemplate=DEBUG
logging.level.org.springframework.jdbc.core.StatementCreatorUtils=TRACE
```

---

[42] *https://docs.spring.io/spring-data/jdbc/docs/current/reference/html/#jdbc.query-methods.named-query*

7.11 Good Design with Repositories

This section aims to discuss how to make the most of Spring Data repositories while following good architectural patterns. However, there are instances where the possibilities provided by Spring and the need for an isolated architecture of separate layers may conflict with each other.

7.11.1 Abstractions through the Onion Architecture

According to Domain-Driven Design, repositories are part of the domain, thus a part of the core in an *onion architecture*. However, they often have a series of technological references. Initially, it's the Spring repository interfaces that are extended, such as `CrudRepository`, `JpaRepository`, or possibly the `*Executor` interfaces. But the problem doesn't end with interfaces. Types such as `Sort` or `Pageable` quickly make their way into the core, or with `QuerydslPredicateExecutor`, proprietary types come from Querydsl. These framework references must, strictly speaking, remain in the infrastructure layer and have no place in the onion core. It's certainly okay to use value objects such as `String` or `LocalDate`, but where do you draw the line?

Another point is the derived query methods. They are actually technology-independent but have their own problems due to their semantic method names. When "magic" method names don't help, `@Query` is specified. The annotation itself has a framework reference, and the string with query language, such as SQL or JPQL, is a clear technological reference. If `@Query` has JPQL or Native SQL, specialties of data modeling from the RDBMS can quickly find their way into the core, but these details have no place there! We may have to reprogram the domain model if something changes in the tables, and that is the wrong direction according to domain-driven design.

Even the entity beans have a technological reference through Jakarta Persistence, as well as mapping information on persistent attributes. Jakarta Persistence works with entity beans, and if the data is to be stored in MongoDB, then they are documents with completely different metadata. The onion core would have to be changed when switching to a different data store.

Of course, we could argue that they are "just" annotations. This perspective is often encountered and acceptable, but if we take domain-driven design seriously, there should be no reason to touch the core when something changes in the infrastructure.

Naturally, attempting to eliminate all technological dependencies from the core is feasible, but it demands significant effort. Typically, developer teams prioritize functionality and agility, especially for software that is of a manageable size, rendering the amalgamation of technologies less problematic in practice.

The more we want to separate the core from the domain logic and the infrastructure, the more duplications arise through interfaces in the onion core and implementations in the infrastructure layer. An inversion of control (IoC) container helps because calls can be abstracted via interfaces in the core so that, at least in the code, the core isn't left behind. Complete separation means that on one side, there is a Java interface, and on the other side, there is an implementation. It can work with `EntityManager`, with Querydsl, or with a JDBC connection, but that happens in the background. The interface in the core of the domain doesn't know how the implementation is realized.

7.11.2 Think of the Interface Segregation Principle and the Onion!

When we declare a repository interface, it should not have too many methods. The `CrudRepository` is extensive, so to reduce the methods, we can extend the base type `Repository`, which first has no methods at all. The next step is to add the methods we want.

It may happen that our own methods in the repository interface make the number of methods large as well. When modeling, we should consider the *Interface Segregation Principle* (ISP): instead of one larger interface—and every client gets access to everything—it's better to write several smaller interfaces. In the smaller interfaces, only the methods that require different classes with business logic are declared.

Consider the following scenario in which a "large" `ProfileRepository` interface is accessed by two clients:

```
public interface ProfileRepository extend JpaRepository<Profile, Long> {

  //
  // Client 1
  //

  // Optional<Profile> findById( Long id );

  @Query(…)
  List<Profile> findByManelength( int manelength );

  //
  // Client 2
  //

  @Query(…)
  long deleteByNickname( String nickname );
}
```

Because `ProfileRepository` inherits from `JpaRepository`, there are about 30 methods. Our own `ProfileRepository` adds two methods: `findByManelength(…)` and `deleteByNickname(…)`. The first client should use two methods: `findById(…)` (from `JpaRepository`) and `findByManelength(…)`. The second client only needs `deleteByNickname(…)`. This is a "good" example of a `ProfileRepository` containing far too many methods, of which clients need less than 10%.

There is a simple solution that satisfies both the ISP and onion architecture principles. The first step is to declare as many interfaces as clients with different usage patterns. Each client gets its individual set of methods:

```
public interface ProfileRepository {          // Client 1
  Optional<Profile> findById( Long id );
  List<Profile> findByManelength( int manelength );
}
public interface DeletingProfileRepository {  // Client 2
  long deleteByNickname( String nickname );
}
```

The `@Query` annotation no longer appears because it would be a reference to technology, and that isn't desired. These two interfaces are pure Java interfaces and relatively abstract, if we disregard the technology reference of the entity bean.

You continue to keep these two interfaces in the onion core and put the implementation or extension of the interface outside into the infrastructure layer. This is where Spring Data comes into play: we write a new interface `SpringProfileRepository` that extends three interfaces: the Spring-specific interface `JpaRepository` and our repositories. It then looks like this:

```
public interface SpringProfileRepository extends
  JpaRepository<Profile,Long>, ProfileRepository, DeletingProfileRepository {
 @Query(…) List<Profile> findByManelength( int manelength );
 @Query(…) long deleteByNickname( String nickname );
}
```

It's possible to wire to the interface type if a client expresses a desire for something, such as `@Autowire` to `ProfileRepository` or `DeletingProfileRepository`, but no longer to `SpringProfileRepository`. If a base type is needed, Spring will always inject the implementation of `SpringProfileRepository`, but this is merely a runtime detail. Therefore, a client only accesses other domain interfaces, and the approach is followed that the outer infrastructure layer can look inside, but not vice versa.

7.11.3 Fragment Interface

While Spring Data offers many options, sometimes operations can't be performed through annotated methods. In certain situations, such as when metadata is evaluated at runtime or when importing CSV files, it may be necessary to use the `EntityManager` or even JDBC connection.

This problem can be solved by using *fragment interfaces*. First, a new interface with additional methods is introduced. Then, the fragment interface is implemented using all Spring-managed beans. Finally, the fragment interface is added to the repository, making the method available to the client.

Step 1: Declare a Fragment Interface

Here's an example involving a fictional `NewRepositoryMethods` interface:

```
public interface NewRepositoryMethods {
  int doSomething( int parameter );
}
```

The interface contains a method. It can have a parameter list and a return.

Step 2: Implement the Fragment Interface

The fragment interface is implemented in the next step. The special feature is that the class name must have a special structure: It must start with the name of the interface (`NewRepositoryMethods`) and end with the suffix `Impl`. The class name is therefore semantic.

This class implements all abstract methods of the interface:

```
public class NewRepositoryMethodsImpl implements NewRepositoryMethods {
  // @Autowired DataSource EntityManager …;
  @Override public int doSomething( int parameter ) {
    return go go go;
  }
}
```

Although an explicit annotation for a Spring-manged bean is missing, something can be injected into the object (as usual with a constructor injection, setter injection, or field injection), and thus the `EntiyManager`, `DataSource`, or whatever service necessary can be accessed. This way, you can go as deep as necessary into the database and implement the operation.

Step 3: The Repository Inherits the Fragment Interface

The last step is to extend the custom repository `NewRepositoryMethods`, that is, the fragment interface:

```
public interface ProfileRepository extends
    JpaRepository<Profile, Long>,
    NewRepositoryMethods { … }
```

The third step involves Spring generating the implementation of our `ProfileRepository` interface at runtime. This can be combined with the Interface ISP so that the client only sees the fragment interface or, even better, another interface that extends the fragment interface.

The fragment interface and its implementations are well-suited for the onion architecture. The fragment *interface* is placed at the core, while the implementation is kept in the infrastructure layer.

Fragment interfaces are especially useful for making changes via Querydsl, which isn't easy with `QuerydslPredicateExecutor`. Furthermore, fragment interfaces are part of Spring Data Commons and are available for all family members, including Spring Data JPA, Spring Data MongoDB, and Spring Data JDBC.

7.12 Projections

There are various reasons why the data from a repository may not be suitable for use with data types in the service layer. One reason could be the presence of specific data types, such as `Page`, which may not be necessary in the service layer. Additionally, if the repository loads data that needs to be filtered and sent through a REST interface, a transformation must be performed. It may also be necessary to reduce the amount of data sent to exclude sensitive information, which can also reduce the data volume.

Another issue is with unsuitable data types. For instance, a profile may reference photo objects, but it may be preferable to only pass the photo names to the outside. In some cases, an entity bean may have insufficient data and require enrichment with additional data. However, modifying the entity bean may not be feasible because certain states may not be required by the client.

When an entity bean is converted to another format, it's called a *projection*. It's essential to perform this transformation to ensure that the data types and formats used in the service layer are appropriate for the client.

7.12.1 Perform Projections Yourself

Projections can be programmed in different ways. In the past, we've realized mappings by hand:

- A JPQL expression can be used to select the desired persistent attributes, and the rest will be ignored. The client can receive the results as a `Map` or as a `Tuple`. In Chapter 6, Section 6.6.9, we discussed `Tuple` for the first time.
- A different approach to retrieving data using JPQL is through *constructor expressions* (see also Chapter 6, Section 6.6.9). By using the `new` keyword in the JPQL query, a new container can be created via a parameterized constructor. This container holds the persistent attributes, and the Jakarta Persistence provider later returns the constructed objects. This variant is very efficient because the results aren't managed entity beans. The constructor expressions provide an alternative to using entity beans and can help to reduce overhead and improve performance when retrieving data.
- Another way to go about projection is to manually convert the repository entity beans into target objects. This is common for controllers, as they transfer the objects (which can be entity beans) into a *data transfer object* (DTO), which is given to the outside by the controller.

7.12.2 Projections in Spring Data

Constructor expressions are only available in Jakarta Persistence and in Spring Data JPA, but not in, for example, MongoDB. However, manual mapping is usually not necessary as the framework can handle the mapping automatically. This is where Spring Data's projections come in handy, as they work across all Spring Data family members and allow for different types of projections.

However, there is a limitation to Spring Data's projections: they only provide a read-only view of the data. It's not possible to transform the data back for saving purposes using the `save(…)` method. In such cases, manual mapping back to the original entity bean is required.

7.12.3 Interface-Based Projection

The interface-based projection is the simplest projection of Spring Data. In the first step, an interface is declared that contains a subset of the getters:

```
public interface SimpleProfile {
  String    getNickname();
  short     getManelength();
  LocalDate getBirthdate();
}
```

Interface `SimpleProfile` allows access to three persistent attributes. It's important that the method names match exactly with those in the original entity bean. While the method names must match, there is some flexibility in the return types. This is because, as in many other places, type conversion is performed with the `ConversionService` (see Chapter 3, Section 3.6), which can convert data types. This means that we could easily return an `int` for the `getManelength()` method, and, of course, custom converters can also be added. This is particularly useful because it allows for any data type to be returned.

Let's look at an example of how this interface-based projection is applied. We have a regular repository:

```
public interface ProfileRepository extends JpaRepository<Profile, Long> {
  // Optional<Profile> findByNickname( String nickname );
  // ->
  Optional<SimpleProfile> findByNickname( String nickname );
}
```

Normally, method `findByNickname(…)` would return `Profile` objects, but the return type can change to `SimpleProfile` or to `Optional<SimpleProfile>`. Thus, the caller gets to see only a subset of the getters. Spring builds a proxy object that implements `SimpleProfile` for this purpose.

Nested Projections

Interface-based projection can also be used for subreferenced entities. Let's take `Profile` as an example. There is a 1:1 relationship between `Unicorn` and `Profile` and a 1:n association between `Profile` and `Photo`. Normally, with our `Profile` entity bean, we would have a method such as `getPhotos()` that returns a collection of `Photo` instances. However, Spring Data can also pass the projections to the children. Here's an example:

```
public interface SimpleProfile {
  // List<Photo> getPhotos();
  // ->

  interface SimplePhoto {
    String getFilename();
  }

  List<SimplePhoto> getPhotos();
}
```

In `SimpleProfile`, there is a nested interface called `SimplePhoto` that only includes the file name of the photo, and not the other details of the `Photo` object, which aren't relevant in this particular example. The `getPhotos()` method returns a list of `SimplePhoto`, indicating that Spring Data also projects the associated elements.

7.12.4 Projections with SpEL Expressions

With the interface-based projection, only subsets are possible. That is, the number of methods only becomes smaller. However, methods with `@Value` and a SpEL expression can be added.

Here's an example: A naw method `formatNicknameAndDescription()` should combine the nickname and the profile description into one string:

```
public interface SimpleProfile {
  // String    getNickname();
  // short     getManelength();
  // LocalDate getBirthdate();

  @Value( "#{target.nickname + ' (' + target.description + ')'}" )
  String formatNicknameAndDescription();
}
```

We've almost always used the `@Value` annotation in the past with a dollar sign and the curly braces to access `Environment` properties. The `@Value` annotation can also be used with SpEL. Then, the notation is `#{…}`. The curly braces contain the actual expression. The predefined variable `target` references the original entity bean. This can be used to refer to `nickname` and `description`. The notation `target.nickname` leads to the getter call of the original entity bean. The SpEL expression in the example continues to add spaces and parentheses.

SpEL expressions are flexible and powerful. In addition, other Spring-managed beans can be referenced with `@` in a SpEL expression. We can rewrite the example from before like this:

```
public interface SimpleProfile {
  // …
  @Value( "#{@profileFormatter.nicknameAndDescription(target)}" )
  String formatNicknameAndDescription();
}
```

The `SimpleProfile` interface still has the `formatNicknameAndDescription()` method, but the program logic is no longer in the SpEL expression; it's offloaded to a Spring-managed `ProfileFormatter` bean:

```
@Component
class ProfileFormatter {
  public String nicknameAndDescription( Profile profile ) {
    return profile.getNickname() + " (" + profile.getDescription() + ")";
  }
}
```

The component has its own `nicknameAndDescription(…)` method that takes a `Profile` object and returns a string with the same structure. The component that is automatically created is called `profileFormatter` (lowercase class name), and the Spring-managed bean can be accessed by the SpEL expression with `@profileFormatter`. The method of the Spring-managed bean is called in the SpEL expression, and `target` is passed, that is, the reference to the original entity bean.

SpEL expressions are convenient, but they have some disadvantages. First, they are expressions, not statement blocks. In addition, type checking isn't guaranteed in strings, and an IDE must already be very good to be able to detect errors. The delegation we see here is useful because it allows us to offload program parts into a real method instead of having them in a string.

7.12.5 Projections with Default Methods

The projection interface can contain default methods. For example, a new default method called `getAgeInYears()` can refer to `getBirthdate()` and calculate the age:

```
public interface SimpleProfile {
  // String    getNickname();
  // short     getManelength();
  LocalDate    getBirthdate();

  default int getAgeInYears() {
    return (int) ChronoUnit.YEARS.between( getBirthdate(),
                                           LocalDate.now() );
  }
}
```

The client gets a new method this way.

7.12.6 Class-Based Projections

So far, we've used interface-based projections where Spring builds a proxy object at runtime. An alternative is to use custom classes with parameterized constructors or records that a repository can also return. Let's take a look:

```
public record SimpleProfile(
    String    nickname,
    short     manelength,
    LocalDate birthdate
) { }
```

Record `SimpleProfile` has three record components, which are named the same as the persistent attributes, that is, `nickname`, `manelength`, and `birthdate`, not `getNickname`, for example!

The repository looks the same as before. Spring will no longer create proxy objects, but our data types, which can, of course, contain other methods.

7.12.7 Dynamic Projections

Up to this point, we've always directly specified our containers, which means that the corresponding repository methods have always returned a subset of the persistent attributes, either through a proxy object or through their own container. However, there is an alternative approach that allows the client to choose which container and format is best for a given context. To achieve this, we need to prepare something on the repository side, which looks something like the following:

```
interface ProfileRepository extends JpaRepository<Profile, Long> {
  <T> Optional<T> findByNickname( String nickname,
                                  Class<T> projection );
}
```

The `findByNickname(…)` method gets a second parameter. The `Class` object determines the container type. The `Class` object has the type `T`, which is what the generically declared method returns. In our case, it's an `Optional<T>`, but, of course, we could have written `T` directly, and then a `null` test would be necessary.

As a selection, we can provide two records:

```
record SimpleProfile(
  String nickname, short manelength, LocalDate birthdate
) { }
record PersonalInfoProfile(
  String nickname, String description
) { }
```

If the client calls the `findByNickname(…)` method, the type token passed will determine the projection target. Let's assume `profiles` is an injected `ProfileRepository`:

```
Optional<SimpleProfile> p = profiles.findByNickname(
                              "FillmoreFat", SimpleProfile.class );
Optional<PersonalInfoProfile> q = profiles.findByNickname(
                              "FillmoreFat", PersonalInfoProfile.class );
```

The use of type tokens provides a high level of flexibility, as the repository method was limited to a specific projection type previously. By using type tokens, the client can now select the target container format as desired. Additionally, if new container formats are introduced in the future, the repository implementation doesn't require modification. This approach allows for the creation of custom containers by any interested party.

Note

There are trade-offs involved in using these solutions. Spring Data projections have a drawback in that they retrieve all the data and then map it to the smaller container. This results in more data being requested than what is required later. This is particularly inconvenient when `EAGER`-loaded associations aren't used. Moreover, when using proxy objects, this can be more disadvantageous because the underlying entity bean is strongly referenced and thus takes up space.[43] In contrast, when working with Jakarta Persistence, the most efficient solution is to use JPQL constructor expressions.

7.13 [Fetchable]FluentQuery *

In addition to the `*Repository` interfaces, we've come to know three `*Executor` interfaces:

- `JpaSpecificationExecutor<T>`
- `QueryByExampleExecutor<T>`
- `QuerydslPredicateExecutor<T>`

They can return entities such as `Optional<T>`, `List<T>`, or `Page<T>`. Notably, the enumeration mentions type `T` because the `*Executor` types are also generically parameterized. Because we want to maximize reusability, we have a small problem whenever we program a `Specification`, `Example`, or `Predicate`, and then want to adjust it afterwards, for instance, through the limitation of results, a different sorting or pagination, or other return types.

Because the `*Executor` interfaces can accept a `Pageable`, this is a solution for sorting and pagination, but if the return of `Page` isn't desired, this variant leads to nothing. There is another solution, however. All three `*Executor` interfaces offer additional methods that accept a transformation function in addition to the `Specification`, `Example`, or `Predicate`, so that the implementations of `Specification`, `Example`, or `Predicate` can be further adjusted afterwards:

```
public interface JpaSpecificationExecutor<T> {
  …
  <S extends T, R> R findBy(
    Specification<T> spec,
    Function<FluentQuery.FetchableFluentQuery<S>, R> queryFunction
  );
}
```

---

[43] *https://github.com/spring-projects/spring-data-commons/blob/main/src/main/java/org/springframework/data/projection/ProxyProjectionFactory.java*

```
public interface QueryByExampleExecutor<T> {
  …
  <S extends T, R> R findBy(
    Example<S> example,
    Function<FluentQuery.FetchableFluentQuery<S>, R> queryFunction
  );
}

public interface QuerydslPredicateExecutor<T> {
  …
  <S extends T, R> R findBy(
    Predicate predicate,
    Function<FluentQuery.FetchableFluentQuery<S>, R> queryFunction
  );
}
```

The transformation function maps an object of type `FluentQuery.FetchableFluentQuery`, to `R`, where `FetchableFluentQuery` is a subtype of `FluentQuery`. The basic structure for the call follows using `Profile` as an example:

```
*Executor.findBy(
  // Specification, Example, Predicate
  …,
  // Function<FluentQuery.FetchableFluentQuery<S>, R>
  (FluentQuery.FetchableFluentQuery<Profile> q) -> …
).all()/oneValue()/one()/first()/firstValue()…
```

A `Function` could look something like this:

```
(FetchableFluentQuery<Profile> q) -> q.sortBy( Sort.by( "nickname" )
```

Spring Data has implementations of the `FetchableFluentQuery` interface for the various `*Executor` types and uses it to call the `apply(…)` method of our `Function` implementation. Our implementation then calls the various `FetchableFluentQuery` methods and thus specifies the request. The methods of `FetchableFluentQuery` include the following, among others:

- `FluentQuery.FetchableFluentQuery<T> limit(int limit)`
- `FluentQuery.FetchableFluentQuery<T> sortBy(Sort sort)`
- `FluentQuery.FetchableFluentQuery<T> project(Collection<String> properties)`
- `default FluentQuery.FetchableFluentQuery<T> project(String... properties)`

You don't come across a `[Fetchable]FluentQuery` that often.

7.14 Auditing *

When accessing databases, it's occasionally important to determine who made a change to an entity and when. The recording of this metadata is called *auditing*. Jakarta Persistence knows `@EntityListeners`, for example. This can be used to add information before writing an entity, for example, for the last access. Of course, we could implement auditing this way manually, but there are better approaches.

7.14.1 Auditing with Spring Data

Spring Data supports simple auditing out of the box. Moreover, this isn't specific to Spring Data JPA but is also supported in other Spring Data family members, including MongoDB.

For auditing, metadata must be written to columns and fields. There are two ways to mark these places:

- **Declaratively via annotations**
  There are `@CreatedBy` ("Who created the data?"), `@LastModifiedBy` ("Who modified the data last?"), `@CreatedDate` ("When was the data created?"), and `@LastModifiedDate` ("When was the data last modified?") annotations.
- **Code implementation of interface `Auditable`[44] or by extending `AbstractAuditable`[45]**
  Interface `Auditable` declares various setters/getters, and `AbstractAuditable` offers the persistent attributes directly. We saw this data type briefly in Chapter 6, Section 6.10.11.

7.14.2 Auditing with Spring Data JPA

Let's look at how auditing can be used in the case of Spring Data JPA.

@EnableJpaAuditing

First, you have to set `@EnableJpaAuditing`[46] to a configuration, such as `@SpringBootApplication`:

```
@SpringBootApplication
@EnableJpaAuditing
public class Data4uApplication { … }
```

---

[44] *https://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/domain/Auditable.html*

[45] *https://docs.spring.io/spring-data/jpa/docs/current/api/org/springframework/data/jpa/domain/AbstractAuditable.html*

[46] *https://docs.spring.io/spring-data/jpa/docs/current/api/org/springframework/data/jpa/repository/config/EnableJpaAuditing.html*

AuditingEntityListener

If Jakarta Persistence is used, the implementation works through a predefined entity listener `AuditingEntityListener`.[47] The task of the `AuditingEntityListener` is to look at the persistent attributes, find out if they are annotated, and, if so, update or set them accordingly.

An entity listener is set via `@EntityListeners`:[48]

```
@Entity
@EntityListeners( AuditingEntityListener.class )
class Photo …
```

The entity, when using annotations, gets the persistent attributes assigned with `@CreatedBy`, `@LastModifiedBy`, `@CreatedDate`, or `@LastModifiedDate`. Here's an example:

```
@Column( nullable = false, updatable = false )
@CreatedDate
private LocalDateTime created;

@Column( nullable = false )
@LastModifiedDate
private LocalDateTime updatedAt;
```

Spring Data auditing allows recording the creation and modification time of entities in persistent attributes. The `@CreatedDate` column is always occupied and hence not nullable. During an update, the `@CreatedDate` column need not be updated because it's only written once during insertion. On the other hand, the `@LastModifiedDate` column will be written several times during the lifecycle and hence can't be `null`. Therefore, setting `updatable = false` for the `@LastModifiedDate` column would be false.

7.14.3 AuditorAware for User Information

Of the four annotation types—`@CreatedBy`, `@LastModifiedBy`, `@CreatedDate`, `@LastModifiedDate`—two have the component "Date", which says that a timestamp is written. The other two annotation types contain "By", which is where a user identifier is written. However, the question is, how does the framework get the current user identifier?

By default, Spring can't automatically determine a user identifier—we have to help. To use `@CreatedBy` and `@LastModifiedBy`, a component of type `AuditorAware<String>`[49] must supply the name. The declaration of the interface looks like this:

---

[47] *https://docs.spring.io/spring-data/jpa/docs/current/api/org/springframework/data/jpa/domain/support/AuditingEntityListener.html*

[48] *https://jakarta.ee/specifications/persistence/3.1/apidocs/jakarta.persistence/jakarta/persistence/entitylisteners*

[49] *https://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/domain/AuditorAware.html*

```
public interface AuditorAware<T> {
  Optional<T> getCurrentAuditor();
}
```

Now let's look at two examples of how a Spring-managed bean of type `AuditorAware` can be created:

- **Example 1:** The username is taken from the system property:

  ```
  @Bean
  AuditorAware<String> systemUserNameAuditorAware( Environment env ) {
    return () -> Optional.ofNullable( env.getProperty( "user.name" ) );
  }
  ```

  The first implementation gets the username from the system properties. On a server, this makes little sense because a username would not depend on the logged-in user.

- **Example 2:** The username is taken from `SecurityContextHolder`:

  ```
  @Bean
  AuditorAware<String> securityContextHolderAuditorAware() {
    return () -> Optional.ofNullable(
      SecurityContextHolder.getContext().getAuthentication().getName()
    );
  }
  ```

  The second variant uses `SecurityContext`, from which you can get an `Authentication` object that returns a name via the `getName()` method. However, it's a prerequisite that an application uses *Spring Security*; otherwise, the data types aren't available. We'll discuss this in Chapter 9, Section 9.17.4.

7.14.4 Outlook: Spring Data Envers

Spring Data auditing has two significant limitations that may not be suitable for real auditing purposes. First, it doesn't provide any information about who made specific modifications, what exactly was changed, and at which time. Although metadata about the logged-in user and the timestamp is available, it doesn't offer sufficient details about the changes made. Secondly, the metadata about users and times is stored alongside the entity. Therefore, if data is deleted, all information is lost.

Although the Spring Data auditing approach is sufficient for displaying information such as when a page was created or last modified, it may not be suitable for real auditing requirements. Fortunately, there are alternative approaches such as the *Hibernate Envers* project[50] (*https://hibernate.org/orm/envers/*) project, which addresses these limitations by using extra tables to store the changed data.

---

[50] Fun fact: Envers is a French word meaning "backside."

The project *Spring Data Envers*[51] integrates Hibernate Envers into the Spring Framework. The process is straightforward, where you need to set the `@Audited` annotation to the entity bean, and the rest is done automatically. Hibernate Envers will create new tables and record who changed which row and when, which includes the entire history of all changes. If any persistent attributes should not be recorded, they can be annotated with `@NotAudited`.

However, the fact that Hibernate Envers writes all changes to tables isn't the only benefit. There is also a revision repository that can be used to query and track the changes made. This is an essential feature for auditing purposes, as it provides a complete history of all changes made to the entities. For detailed information, it's recommended to refer to the reference documentation of the Hibernate site.

Note

When all changes to rows are managed on the software side, the costs can increase significantly because every modification automatically triggers a backup of the data in the revision tables in the background. Professional database management systems typically provide native auditing capabilities, which can make the auditing process more efficient because the code and tables are closely integrated with the database management system. However, Hibernate Envers has a notable advantage in terms of portability because it can work with any RDBMS. This means that it's not tied to a specific database management system and can be used across multiple platforms, making it a flexible and adaptable auditing solution.

7.15 Incremental Data Migration

When dealing with conventional databases, it becomes apparent that the data can be significantly older than the software that accesses it. Software often undergoes frequent changes due to evolving programming languages and libraries. In contrast, database data remains consistent over long periods. For instance, the Berenberg Bank, which is one of the oldest banks in Germany, has been recording data for more than 400 years, but it no longer uses pen and paper. The format of the data and the type of information collected have undoubtedly changed over time, with some data becoming redundant, while other data is now more critical. There has also been a need to migrate data to more modern formats as technology advances.

This section focuses on the incremental migration of databases from one version to the next. This is a crucial aspect of managing databases, given the historical nature of the data and the need to keep up with changing technological trends.

---

[51] *https://spring.io/projects/spring-data-envers*

7.15.1 Long Live the Data

Refactoring software is nothing unusual, and it's also an important operation for databases. The data organization that was optimal in the past may not be so today. This may be due to various reasons:

- Data types can change.
- Columns or tables are to be renamed, for example, by integrating them into another language area.
- Data can be normalized or denormalized, and data can be distributed to different tables or transferred from different tables back to one table for performance reasons.
- In the case of a company merger, different databases may be merged. Most likely the data of the different companies has been modeled differently, so a normalization of the data must inevitably take place.
- The database management system may change, requiring adjustments, for example, in the choice of data types or stored procedures.

Agile programming teams are familiar with regular refactoring and understand the importance of staying current with new requirements. However, database structures are often not adapted, and software may need to work with outdated databases, leading to inefficiencies.

7.15.2 Evolutionary Database Design

The process of regularly adapting both database schema and software to meet evolving needs is known as *evolutionary database design*. This involves making incremental improvements to the database schema, gradually modernizing it step by step. *Migration scripts* are commonly used in practice to achieve this. These scripts gradually transfer the database from a version 1 schema to a version 2 schema. Typically, these migration scripts are SQL scripts that can be applied to the database by a special tool or by the user's own software. For example, at startup, the software could first check the current version and then update the database, if necessary, before starting work. This approach ensures that the software always works on the most current schema version.

Handling different database schemas simultaneously increases the complexity of software. A switch in the software is needed to determine which program code to use, depending on the schema. This can lead to increased scope and maintenance requirements for the software if it has to handle different versions. On the other hand, if the software only needs to consider one state of a database schema at a time, it becomes simpler and leaner to manage.

In the realm of Java, there are two well-known solutions for database migration:

- *Liquibase* (*www.liquibase.org*)
- *Flyway* (*http://github.com/flyway/flyway*)

The two open-source tools, Liquibase and Flyway, are both licensed under Apache. Although both are popular, Liquibase is gaining more traction.[52]

Flyway supports pure SQL scripts and Java code, while Liquibase uses XML files that are then converted to SQL. Liquibase is a better fit for environments that use different database management systems, which is common in cloud-based systems. For instance, a free, open-source database may be used during development, while a commercial database may be used in the production system. Liquibase translates declarative XML files into SQL for the appropriate database management system. Flyway is a bit simpler to use than Liquibase.

7.15.3 Incremental Data Migration with Flyway

Flyway carries out *migrations* via SQL scripts or Java code. Each of these migrations has a version number, for example, *V1*, *V2*, or *V2.3*. Flyway sorts the migrations according to the version number and imports them into the database. In the database, the current schema is stored, as well as the migrations performed in the past. This means that Flyway starts with the lowest version number and works its way up to the highest version number. If Flyway finds new migrations that aren't recorded in the migration table, they are imported.

Flyway in Spring Boot: Dependency

Flyway is integrated well with Spring Boot. Only one dependency is needed.

```
<dependency>
  <groupId>org.flywaydb</groupId>
  <artifactId>flyway-core</artifactId>
</dependency>
```

Listing 7.5 pom.xml Extension

We'll see in a moment that Maven can also perform the migrations—then further entries in the POM file would be necessary. But in our case, Flyway is supposed to adapt the database as part of the Spring application.

7.15.4 Flyway in Spring Boot: Migration Scripts

It has been mentioned that migrations can be in the form of either SQL scripts or Java programs. For now, let's focus on SQL scripts. These files come in the classpath by default. Also by default, the source files are located in the *db/migration* subdirectory under *src/main/resources/*. However, it's possible to customize the location by using a configuration property. This can be helpful in using different migration scripts for different databases.

---

[52] To estimate the popularity, *https://stackshare.io/stackups/flyway-vs-liquibase* gives an orientation.

Next, we put the SQL migration scripts into the directory. The file names have a special structure:

```
<Prefix><Version>__<Description>.sql
```

The name of the script starts with a prefix, followed by the version number, then two underscores, and some descriptive text.

To illustrate how this can appear, let's examine three sample file names:

- *V1_0__Create_all_tables.sql*
- *V1_1__Add_foreign_keys_references.sql*
- *V2_0__Insert_initial_data.sql*

Flyway follows a convention-over-configuration approach and supports various database management systems. The prefix `V` stands for a versioned SQL script. These scripts are named with a version number separated by underscores instead of periods, followed by a brief description.

The versioned SQL scripts play a crucial role in Flyway. For example, a first script could build tables, the second script to set foreign key references, and the third script to insert initial data into the database. Nevertheless, the content of these scripts varies based on the project's specifications. Developers may need to modify tables, add new columns, transfer components between tables, or enact other alterations as necessary.

Flyway has three different prefixes for scripts, and `V` is one of them. In addition to `V`, there is `U` for *Undo* scripts and `R` for *Repeatable* scripts. Developers use Undo scripts to roll back changes made to the database, while Repeatable scripts are used to update stored procedures. Unlike versioned scripts, the checksum of Repeatable scripts can change. Whenever Flyway detects a change to these scripts, it will re-import them into the database.

Flyway uses checksums to detect changes made to scripts. Versioned scripts must not change their checksum, or it will cause issues. All of this information about scripts, including their version numbers and descriptions, is stored in the metatable.

Evolutionary Database Design in the Date4u Database

With the *Date4u database*, we can easily use Flyway. In the first step, we delete the database. This can be done with the SQL statement `DROP ALL OBJECTS`, for example, from the H2 console.

Next, we need to create the new *db/migration* directory. We want to create the previous Date4u database using three SQL scripts that we've already briefly learned about:

- *V1_0__Create_all_tables.sql*
- *V1_1__Add_foreign_keys_references.sql*
- *V2_0__Insert_initial_data.sql*

We distribute the original monolithic file *unicorns.sql* (from *https://tinyurl.com/4fu3hwu4*) among the three scripts. After that, if we start the program, the logging will show the application of the migration:

```
… o.f.core.internal.command.DbValidate     : ↩
Successfully validated 3 migrations (execution time 00:00.014s)
… o.f.c.i.s.JdbcTableSchemaHistory         : ↩
Creating Schema History table "PUBLIC"."flyway_schema_history" …
… o.f.core.internal.command.DbMigrate      : ↩
Current version of schema "PUBLIC": << Empty Schema >>
… o.f.core.internal.command.DbMigrate      : ↩
Migrating schema "PUBLIC" to version "1.0 - Create all tables"
… o.f.core.internal.command.DbMigrate      : ↩
Migrating schema "PUBLIC" to version "1.1 - Add foreign keys references"
…. o.f.core.internal.command.DbMigrate     : ↩
Migrating schema "PUBLIC" to version "2.0 - Insert initial data"
… o.f.core.internal.command.DbMigrate      : ↩
Successfully applied 3 migrations to schema
```

A look into the database shows a new table `flyway_schema_history` (see Table 7.3).

installed_rank	–1	1	2	3
version	NULL	1.0	1.1	2.0
description	<< Flyway Schema History table created >>	Create all tables	Add foreign keys references	Insert initial data
type	TABLE	SQL	SQL	SQL
script		V1_0__Create_all_tables.sql	V1_1__Add_foreign_keys_references.sql	V2_0__Insert_initial_data.sql
checksum	NULL	-1593638494	1007985003	1243822560
installed_by	USER	USER	USER	USER
installed_on	2022-07-20 21:06:30.687000	2022-07-20 21:06:30.711000	2022-07-20 21:06:30.748000	2022-07-20 21:10:32.736000
execution_time	0	7	26	19
success	true	true	true	true

Table 7.3 “flyway_schema_history” Table

Note

Flyway must find an empty database at the beginning. If tables exist in the database, Flyway reports an error, even if, for example, there is a `DROP ALL OBJECTS` in the first line of the first SQL script.

Solution: We set the switch `spring.flyway.baseline-on-migrate` to `true`. Then, Flyway would create its metadata table even if other tables already exist with data. There are a number of other configuration properties listed at *https://docs.spring.io/spring-boot/docs/current/reference/html/application-properties.html#appendix.application-properties.data-migration*.

7.15.5 Migrations in Java Code

Certain incremental database updates aren't possible through SQL scripts:

- Converting types and format changes
- Adding information
- Importing data from CSV files or NoSQL databases

Because this can't be done with regular SQL, migrations can also be programmed in Java code. For this purpose, the `JavaMigration` interface must be implemented. Usually, we fall back to the abstract implementation of the interface, `BaseJavaMigration`,[53] which extracts the prefix and the version from the class name, for example.

Schematically, it looks like this:

```
package db.migration;

import org.flywaydb.core.api.migration.BaseJavaMigration;
import org.flywaydb.core.api.migration.Context;
import …

public class V1_1__Add_Name_Column_to_Customers
    extends BaseJavaMigration {

  @Override
  public void migrate( Context context ) throws Exception {
    try ( Connection con = context.getConnection() ) {
      JdbcTemplate template = new JdbcTemplate(
          new SingleConnectionDataSource(con, true) );
      …
    }
  }
}
```

---

[53] *https://javadoc.io/doc/org.flywaydb/flyway-core/latest/org/flywaydb/core/api/migration/BaseJavaMigration.html*

The class will be placed in package `db.migration`. The naming follows the naming convention of the SQL migration scripts. This may look strange for class names, and a code checker would likely complain about it, but it's a necessary naming convention.

In our case, it's version number 1.1 and a versioned script. From the base type, the `migrate(…)` method must be implemented. When executed, Flyway passes a `Context` object that can be used to query the SQL connection. Then, the code can do everything that needs to be done with the database, including establishing connections to other databases and importing data from there into its own database. Higher level objects can be built with the JDBC connection, such as a `JdbcTemplate` in the example.

7.15.6 Flyway Migrations outside Spring

Flyway automatically processes the migrations when a Spring Boot application is started. Migrations can also be triggered via Flyway using a command-line tool, such as Maven or Gradle, and versions of the database can be viewed.

In the case of Maven, the database connection is entered into the POM file. Then the following can be executed:

- **`mvn flyway:info`**
  Displays the migration table.
- **`mvn flyway:migrate`**
  Performs the migration.
- **`mvn flyway:migrate -dryRunOutput=dryrun.sql`**
  On a dry run, Flyway will generate an SQL script. We can look at this in terms of whether the merged SQL script would work.

More information is available at *https://documentation.red-gate.com/fd/quickstart-maven-184127578.html*; it doesn't take long to read through it, either. The Java migrations are explained at *https://documentation.red-gate.com/fd/tutorial-java-based-migrations-184127624.html*.

Note

It's convenient when a Jakarta Persistence provider creates and changes tables. Hibernate allows incremental improvements up to a point. But with Flyway, it's risky if two tools change the database. Therefore, with Hibernate, the mode should be set to `validate` or `none`:

```
spring.jpa.hibernate.ddl-auto=validate
```

This leaves Flyway as the only tool. For test cases, a database must have the identical schema as the production system, and Flyway helps ensure that the test databases are built the same way as the production databases.

7.16 Test the Data Access Layer

In the preceding sections, we discussed the development of code for accessing databases. Initially, this was accomplished at a granular level using handwritten SQL, which was subsequently abstracted through the use of `EntityManager`, and eventually further simplified with the introduction of repositories. As we move closer to the lower layers, the potential for errors increases, underscoring the importance of thoroughly testing database access code.

7.16.1 What Do We Want to Test?

Database access layers, which contain code for database operations, should be tested just like any other code. However, unlike regular components that undergo *unit testing* with mocked collaborating components, database tests are *integration tests* that operate with a real database in the background. Although it's possible to mock the JDBC connection or repository, it would not be useful because we want to test the repository itself. The repository should only be mocked if a service that accesses the repository needs to be tested.

Tip

When creating database tests, it's crucial to stay focused and avoid testing the database technology or O/R mapper itself. It's not the responsibility of tests to ensure the functionality of the database or whether the O/R mapper is doing its job correctly. Instead, the goal is to verify whether our methods are producing the expected results and to detect potential issues, such as swapped parameters or incorrect conversions.

Spring Data JPA itself tests some things that we would not need to test:

- When Spring Boot applications are launched, it ensures that all derived query methods are valid. For example, if a persistent attribute has been renamed in the entity bean, Spring Boot reports this at startup time. In the IDE, the errors are generally noticeable as well.
- Likewise, Spring Data JPA performs a syntactic check of all JPQL expressions in the `@Query` strings at startup. However, this is only a syntactic check, a semantic check can't be performed. Native SQL queries can't be checked for syntactical correctness because they are heavily dependent on the database management system.

Regardless, however, there is still a lot of work to be done. Spring Data has very extensive testing support, which we'll take a look at.

7.16.2 Test Slices

In the past, we usually annotated tests with `@SpringBootTest`. In this case, the entire container is booted, and all components are initialized. If you recall the *Controller → Service → Repository* chain, there is nothing "behind" the repository anymore. The controller delegates the execution of business logic to a service, and should the business logic request data, it turns to the repository. But because the repository has no relationship to the service, much less to any controller, an `@SpringBootTest` builds up too many components.

Spring Boot provides a number of special test slice annotations, as already explained in Chapter 3, Section 3.9. As a reminder, a test slice only boots a portion of the components, and for database code, only the repositories would need to be initialized.

There are different test slices for testing repositories:

- `@JdbcTest`
- `@DataJpaTest`
- `@DataJdbcTest`
- `@DataR2dbcTest`
- `@DataCassandraTest`
- `@DataCouchbaseTest`
- `@DataElasticsearchTest`

A general overview is provided at *https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#test-auto-configuration*.

@DataJpaTest

All these annotations replace `@SpringBootTest`, and then only a small subset of components is built. Let's stay with `@DataJpaTest`,[54] which does the following:

- Leads to the loading of all configurations, such as from *application.properties* because connection data is often stored there
- Scans and searches all entities
- Initializes `DataSource`, `JdbcTemplate`, `NamedParameterJdbcTemplate`, and `EntityManager`
- Detects and initializes all Spring Data JPA repositories
- Builds an in-memory database under certain circumstances

Other components or factory methods aren't built and called. We can resort to `@Import` if additional components are to be integrated. For example, if the test case wants to use utility components, we write the following:

---

[54] *https://docs.spring.io/spring-boot/docs/current/api/org/springframework/boot/test/autoconfigure/orm/jpa/DataJpaTest.html*

```
@DataJpaTest
@Import( { Service1.class, Service2.class } )
class …
```

7.16.3 Deploy an In-Memory Test Database

There are database management systems programmed in Java, such as H2, Derby, or HSQLB. They can run as embedded databases as part of their own application. If the embedded databases are set in such a way that they don't persist but only hold the data in memory, they are referred to as *in-memory databases*.

Spring Data has a useful feature where an in-memory database is automatically created in the test case if H2, Derby, or HSQLB is present in the classpath. This default connection is then configured with the in-memory database:

```
… beddedDataSourceBeanFactoryPostProcessor : ↩
Replacing 'dataSource' DataSource bean with embedded version
… o.s.j.d.e.EmbeddedDatabaseFactory        : ↩
Starting embedded database: url= ↩
'jdbc:h2:mem:64ab28f2-3d00-4c91-b5d2-37b7bdf11e74; ↩
DB_CLOSE_DELAY=-1;DB_CLOSE_ON_EXIT=false', username='sa'
2
```

The following test case shows that with H2 in the classpath, all references are `non-zero`:

```
@DataJpaTest
class ProfileJpaRepositoryTest {

  @Autowired private DataSource        dataSource;
  @Autowired private JdbcTemplate      jdbcTemplate;
  @Autowired private EntityManager     em;
  @Autowired private ProfileRepository profiles;

  @Test
  void datasource_jdbctemplate_entitymanager_repository_not_null() {
    assertThat( dataSource ).isNotNull();
    assertThat( jdbcTemplate ).isNotNull();
    assertThat( em ).isNotNull();
    assertThat( profiles ).isNotNull();
  }
}
```

7.16.4 Assign Connection to the Test Database

Typically, large Java programs don't use purely Java-based databases. Instead, they often use regular JDBC drivers, such as those for an Oracle database in the classpath.

However, if there is no Java-based RDBMS in the classpath, then an in-memory database won't be created, and the JDBC connection won't be established. In this case, we must carefully consider which test database the test case should connect to.

For a test, a custom database connection can be set explicitly, but if a Java database is in the classpath, Spring Boot anticipates that. Therefore, another annotation is needed in addition to `@DataJpaTest`: `@AutoConfigureTestDatabase( replace = AutoConfigureTestDatabase.Replace.NONE )`.[55] The assignment of `replace` means that Spring Boot should not replace the JDBC URL with its own test database in the test case, but use the JDBC URL we specify. `Replace.NONE` is to be interpreted in such a way that Spring Boot should *not* replace our connection with the in-memory database.

The connection data to the database must be specified via the usual properties, which can also be done differently if, for example, you specify a `DataSource` for the connection yourself, but we can ignore that. Via the `@TestPropertySource` JDBC URL, the username and password can be determined:

```
@DataJpaTest
@AutoConfigureTestDatabase(replace=AutoConfigureTestDatabase.Replace.NONE)
@TestPropertySource( properties = { "spring.datasource.url=…" } )
class …
```

In the case of multiple database tests, it may be useful to keep the connection data externally in a property or YAML Ain't Markup Language (YAML) file. Therefore, approach number two is to create a profile file with the configuration properties for the JDBC URL, username, and password. If the profile is called *test-database*, such a file would be called *application-test-database.properties* and could start with the following lines:

```
spring.test.database.replace=none
spring.datasource.url=…
…
```

The configuration property `spring.test.database.replace=none`[56] has the same effect as `@AutoConfigureTestDatabase(replace=AutoConfigureTestDatabase.Replace. NONE)`.

In the test case, the profile is activated through `@ActiveProfiles` explained in Chapter 3, Section 3.2.12, which leads to the loading of this properties file and configurations.

```
@DataJpaTest
@ActiveProfiles( "test-database" )
class …
```

---

[55] *https://docs.spring.io/spring-boot/docs/current/api/org/springframework/boot/test/autoconfigure/jdbc/AutoConfigureTestDatabase.html*

[56] *https://docs.spring.io/spring-boot/docs/current/reference/html/application-]properties.html#appendix.application-properties.testing*

The second variant is useful when several classes need the same database information. Enumerating the connection data individually via `@TestPropertySource` in several files is inflexible; one file simplifies a change of the connection data in a central place.

7.16.5 Build Tables with Initialization Scripts

When connecting to the test database, the auto-configuration may establish a connection to an in-memory database automatically, or we may need to provide the JDBC connection details ourselves. Once the database connection is established, we next need to determine what tables exist in the database. If we're using a predetermined test database, then we can prepare the necessary tables. However, an in-memory database is always empty at the start.

Assuming that the database is empty and doesn't have any tables, there are four different approaches to create the tables:

- Database migration tools, such as Flyway or Liquibase, can create the tables. If Flyway is used as a tool, care should be taken that a Jakarta Persistence provider doesn't also build the tables. Therefore, the assignment with `spring.jpa.hibernate.ddl-auto=validate` makes sense.
- If an SQL script named *schema.sql* exists in the classpath, Spring Boot will automatically import this file into the database.
- The Jakarta Persistence provider can build the tables. We haven't wanted this in the past because we wanted control over these tables on the database side. In a test case, building can be handy.
- The tables are built manually via the JDBC connection. In principle, this is possible, but it's laborious and therefore rarely useful.

Flyway will also run in the test case, and the log output shows that the migration scripts will also be imported into the test database. So, the test case won't give any error regarding missing tables:

```
@Test
void database_has_tables() {
  List<Profile> profiles = em.createQuery(
    "SELECT p FROM Profile p", Profile.class ).getResultList();
}
```

Note

If you have Flyway in the classpath (e.g., because you use it productively) but don't want to use it in testing, then you would set property `spring.flyway.enabled` to `false`.

Java Persistence API Provider Building Tables

Because we use Flyway, the migration scripts are imported into the test database. This is convenient. If you don't use migration software, you can fall back on the persistence provider, which can also build tables. In the past, we had `spring.jpa.hibernate.ddl-auto` set to `validate`, and that's what needs to be changed so that the configuration property is set to `create-drop`.

Via `@TestPropertySource`, this can be changed for the test case. For Hibernate, it looks something like this:

```
@TestPropertySource( properties="spring.jpa.hibernate.ddl-auto=create-drop" )
```

Then, Hibernate would automatically create the tables in our test database.

Tip

At first glance, the persistence provider seems to build everything completely. However, there are two problems:

- The tables are built based on the metadata in the entity bean. But these might not be complete, for example, concerning `NULL` or the column width.
- There are many additional things in the database, such as stored procedures, triggers, or other constraints that an entity bean can't express. Those who use Flyway or Liquibase should stick with it.

7.16.6 Testcontainers Project

When it comes to the test database, we typically have two options: an in-memory database or specifying a JDBC URL to an existing database. It's crucial to use the same RDBMS in the test case as in production. Although using PostgreSQL in production and H2 for testing may seem tempting for performance reasons, it's not recommended because the SQL syntax may differ between the two databases. For example, H2 supports `TINYINT`, but PostgreSQL only recognizes `SMALLINT`.

Now, let's set aside the H2 database for a moment and assume that the project uses PostgreSQL. In that case, the test case would also need to connect to a prepared PostgreSQL database. However, because test cases should always run automatically, the challenge is how to start the database for the test run.

Thankfully, there is a solution in the form of the *Testcontainers* project (*www.testcontainers.org*). This project allows for launching a Docker container for infrastructure services such as database management systems or email servers. An integration with JUnit starts the Docker container in the background, making it easy to start and connect to the test database seamlessly.

Let's take a look at an example of how, in principle, a PostgreSQL database can be accessed in a Docker container. First, two dependencies are added to the POM file:

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-testcontainers</artifactId>
  <scope>test</scope>
</dependency>

<dependency>
  <groupId>org.testcontainers</groupId>
  <artifactId>junit-jupiter</artifactId>
  <scope>test</scope>
</dependency>

<dependency>
  <groupId>org.testcontainers</groupId>
  <artifactId>postgresql</artifactId>
  <scope>test</scope>
</dependency>

<dependency>
  <groupId>org.postgresql</groupId>
  <artifactId>postgresql</artifactId>
  <scope>test</scope>
</dependency>
```

A dependency on the *Testcontainers* project is always necessary and then, depending on the infrastructure service, a special dependency, in our case for PostgreSQL. Testcontainers supports other infrastructure services, such as messaging systems. New in Spring Boot 3.1 is `org.springframework.boot:spring-boot-testcontainers`, which frees us from some configuration work. Since the test case requires the JDBC driver for PostgreSQL, it is also included in the dependencies—in practice, presumably not in the scope test.

In the next step, JUnit (*www.testcontainers.org/test_framework_integration/junit_5*) can start the container. The JDBC connection data can be queried, and the configuration properties can be set:

```
@DataJpaTest
@AutoConfigureTestDatabase(replace = AutoConfigureTestDatabase.Replace.NONE)
@TestPropertySource(properties = "spring.jpa.hibernate.ddl-auto=create-drop")
@TestPropertySource(properties = "spring.flyway.enabled=false")
@Testcontainers
class TestContainerDemoTest {
  @Container
  @ServiceConnection
  static PostgreSQLContainer<?> __=new PostgreSQLContainer<>(/*Image name*/);

  @Autowired private ProfileRepository profiles;

  @Test
  void test_container_database_exists_but_is_empty() {
    Assertions.assertThat( profiles.count() ).isZero();
  }
}
```

The first annotation, `@AutoConfigureTestDatabase`, tells Spring Boot that we want to determine the JDBC connection data ourselves. In the given example, we intend to use Hibernate as the Jakarta Persistence provider to generate tables. As a result, the `@TestPropertySource` annotation sets the `ddl-auto` switch to `create-drop`. If Flyway is in the classpath, Flyway should not build the tables and the configuration property `spring.flyway.enabled` should be set to `false`. (Flyway is a good idea and would be used in practice, but our SQL is H2-specific and does not work under PostgreSQL).

The first new annotation is `@Testcontainers`. The annotation isn't mandatory, but it shortens the code a bit. `@Testcontainers` leads to searching for `@Container`-annotated instance variables, and the containers are started later. Containers can also be started manually in a `static` block; then you wouldn't need the `@Testcontainers` annotation, but why write more than necessary? We initialize the `@Container`-annotated variable with a new object; in the constructor an image with a version number can also be named.

JUnit will start a Docker container with the database management system in the background. However, the test program does not know the JDBC connection data, so `@ServiceConnection` is used, which reads the connection data to the container and transfers it to the configuration properties spring.datasource.* for the test case.

Thereafter, the test cases can be programmed as before, and `DataSource`, `JdbcTemplate`, and so on are attached to the database in the Docker container. For more details, see the reference documentation at *https://docs.spring.io/spring-boot/docs/current/reference/html/features.html#features.testing.testcontainers.at-development-time*.

7.16.7 Demo Data

To summarize the previous steps for a test case, we first need to establish a connection to the database. This can be achieved by choosing between an in-memory database, accessing running databases with JDBC connection data, or starting a database in a Docker container specifically for the test case.

The next step involves creating tables for the test case, which can be accomplished using Flyway/Liquibase, the Jakarta Persistence provider, a *schema.sql* script, or a custom program.

Test cases for data access code follow the same basic structure as other test cases: a setup is performed, the operation is triggered, and then the result is verified. For the setup phase, demo data is usually required, which can come from various sources.

- The test case can store data itself via the `DataSource`, the `EntityManager`, or the repository.
- In addition to the *schema.sql* file, you can specify a *data.sql* file with global test data to be imported. When Spring runs the test case, the *data.sql* script is loaded and imported into the database.
- All the test methods can be named via `@Sql` corresponding SQL scripts, which are played locally for this method into the database.

7.16.8 @Sql and @SqlGroup

An example of the annotation `@Sql`[57] might look like this:

```
@test
@Sql( "create-ten-demo-unicorns.sql" )
void test() …
```

The `@Sql` annotation is set in addition to the `@Test` annotation to the respective test method and determines which file is to be imported. In our example, the file *create-ten-demo-unicorns.sql* is located in the classpath. Because the `@Sql` annotation is repeatable, multiple files can be specified.

Here's another example: *script1.sql* and *script2.sql* are to be submitted to the database:

```
@Test
@Sql( "script1.sql" )
@Sql( "script2.sql" )
void test() …
```

The annotation `@Sql` is repeatable since Java 8; before that, there were no repeatable annotations. In older program code, you can therefore find the annotation `@SqlGroup`, which contains several SQL scripts as a container:

```
@Test
@SqlGroup( {
  @Sql( "script1.sql" ),
```

---

[57] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/test/context/jdbc/Sql.html*

```
  @Sql( "script2.sql" )
} )
void test() …
```

In modern code, however, this annotation should no longer be found.

7.16.9 TestEntityManager

In package `org.springframework.boot.test.autoconfigure.orm.jpa`, there is a small type called `TestEntityManager`,[58] which Spring builds in the test case and which can be injected. The `TestEntityManager` contains a subset of the methods of `EntityManager` and adds methods that an `EntityManager` doesn't have but are useful for test cases:

- `void clear()`
- `void detach(Object entity)`
- `<E> E find(Class<E> entityClass, Object primaryKey)`
- `void flush()`
- `Object getId(Object entity)`
- `<T> T getId(Object entity, Class<T> idType)`
- `<E> E merge(E entity)`
- `<E> E persist(E entity)`
- `<E> E persistAndFlush(E entity)`
- `Object persistAndGetId(Object entity)`
- `<T> T persistAndGetId(Object entity, Class<T> idType)`
- `<E> E persistFlushFind(E entity)`
- `<E> E refresh(E entity)`
- `void remove(Object entity)`
- `EntityManager getEntityManager()`

For example, two new methods are `persistFlushFind(…)` and `persistAndGetId(…)`. They are useful because they combine two things: saving an entity bean and reloading it directly. This allows you to check in a one-liner if what was written and what comes back is consistent. `persistAndGetId(…)` writes the record and returns only the ID.

7.17 Summary

Spring Data greatly simplifies our work by providing easy access to storage systems. In this chapter, you've learned how Spring Data implements repositories and handles data store operations in the background. As a specific example, we worked with Spring Data JPA, one of the most important members of the Spring Data family.

---

[58] *https://docs.spring.io/spring-boot/docs/current/api/org/springframework/boot/test/autoconfigure/orm/jpa/TestEntityManager.html*

However, because there are alternative storage options beyond RDBMSs, the next chapter will demonstrate how the same mechanisms can be used with a document store and a text search engine.

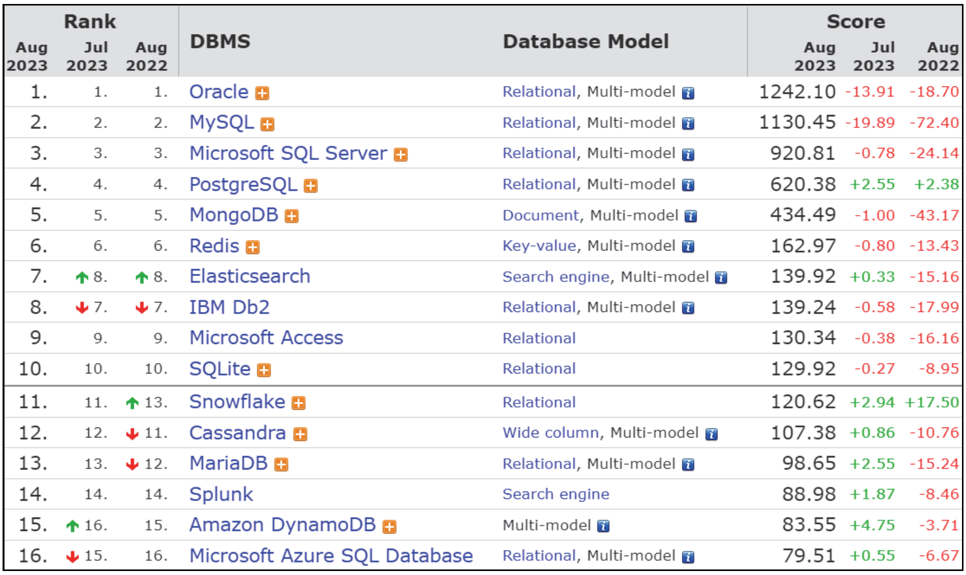
Chapter 8

Spring Data for NoSQL Databases

Although relational database management systems (RDBMS) have been the go-to option for many years, NoSQL databases have gained popularity in recent years due to their flexibility, scalability, and performance. In this chapter, we'll focus on two types of NoSQL databases: document stores and text search engines. We'll discuss the differences between these systems and RDBMSs, and you'll learn how to use them with Spring Data. By the end of this chapter, you'll have a deeper understanding of when to use NoSQL databases and how to integrate them with your Spring applications.

8.1 Not Only SQL

Relational databases use tables for data storage. Despite the emergence of various alternatives, RDBMSs still hold a dominant position in popularity. For a preliminary evaluation, the DB-Engines website at *https://db-engines.com/en/ranking* provides useful insights (see Figure 8.1).



Rank			DBMS	Database Model	Score		
Aug 2023	Jul 2023	Aug 2022			Aug 2023	Jul 2023	Aug 2022
1.	1.	1.	Oracle	Relational, Multi-model	1242.10	-13.91	-18.70
2.	2.	2.	MySQL	Relational, Multi-model	1130.45	-19.89	-72.40
3.	3.	3.	Microsoft SQL Server	Relational, Multi-model	920.81	-0.78	-24.14
4.	4.	4.	PostgreSQL	Relational, Multi-model	620.38	+2.55	+2.38
5.	5.	5.	MongoDB	Document, Multi-model	434.49	-1.00	-43.17
6.	6.	6.	Redis	Key-value, Multi-model	162.97	-0.80	-13.43
7.	↑ 8.	↑ 8.	Elasticsearch	Search engine, Multi-model	139.92	+0.33	-15.16
8.	↓ 7.	↓ 7.	IBM Db2	Relational, Multi-model	139.24	-0.58	-17.99
9.	9.	9.	Microsoft Access	Relational	130.34	-0.38	-16.16
10.	10.	10.	SQLite	Relational	129.92	-0.27	-8.95
11.	11.	↑ 13.	Snowflake	Relational	120.62	+2.94	+17.50
12.	12.	↓ 11.	Cassandra	Wide column, Multi-model	107.38	+0.86	-10.76
13.	13.	↓ 12.	MariaDB	Relational, Multi-model	98.65	+2.55	-15.24
14.	14.	14.	Splunk	Search engine	88.98	+1.87	-8.46
15.	↑ 16.	15.	Amazon DynamoDB	Multi-model	83.55	+4.75	-3.71
16.	↓ 15.	16.	Microsoft Azure SQL Database	Relational, Multi-model	79.51	+0.55	-6.67

Figure 8.1 Popularity of Databases (Source: https://db-engines.com/en/ranking)

Oracle Database is at the top, followed by *MySQL* (also from Oracle since 2010), then *Microsoft SQL Server*, and then comes *PostgreSQL*—they are all RDBMS.

RDBMSs store data in tables, but there are alternative storage structures such as hierarchical documents, key-value pairs for caching systems, and graph-oriented databases. These nonrelational structures are collectively referred to as *NoSQL databases*. However, the term isn't always used consistently:

- NoSQL can stand for *No SQL*, which means no SQL at all. (The term could also stand for alternative query languages for relational databases, but this is rather rare.) Usually, NoSQL means that it's not a relational database and consequently no SQL is used.
- However, it can also stand for *Not only SQL*. That means we have various database management systems, including SQL and others such as MongoDB, Elasticsearch for text search, and so on.

Figure 8.1 shows that the RDBMS at the top is followed by *MongoDB* as the first representative of a NoSQL database, followed by *Redis* and *Elasticsearch*, as further representatives of NoSQL databases.

8.2 MongoDB

MongoDB belongs to the class of *document-oriented NoSQL databases*. In the following, we'll install MongoDB and use a Spring program to access the data in read and write modes.

8.2.1 MongoDB: Documents and Collections

When you hear the term "document," you might think of a word processor, but in our context, you can think of a document as nested JavaScript Object Notation (JSON) objects that have *fields*, which are actually key-value pairs. A document might look like this:

```
{
  "title": "earplugs",
  "description": "reducing the noise in style",
  "category": "thing",
  "price": 400,
  "packaging": {
    "itemQuantity": 10
  }
}
```

A product has a title, description, category, price, and perhaps packaging information.

In MongoDB, documents are grouped in *collections*; symbolically, it looks like this:

```
{ "title": "earplugs", "category": "thing", "price": 400 }
{ "title": "love balls", "category": "thing", "price": 4900 }
{ "title": "Coffee", "price": 9900, "flavor": ["Peach","Cherry"] }
```

The example clearly illustrates that each of the three documents has a slightly different structure, and this is acceptable. One of the main benefits of NoSQL databases is their flexibility in terms of structure. In the Java world, we're familiar with the term `Collection`, which is a data structure that can theoretically hold any objects. A MongoDB collection can be thought of as a persistent `java.util.Collection`.

8.2.2 About MongoDB

The MongoDB database is the most widely used NoSQL database. The RDBMS system is developed in C++ by an American company of the same name, but there are drivers for all kinds of programming languages: Java, C#, Python, and so on. MongoDB is under a special license called the *Server Side Public License* (SSPL).[1]

Schema Freedom

Document-oriented databases such as MongoDB offer the flexibility of *dynamic schema*, where each collection can contain entirely different objects. This means that the structure of the documents doesn't need to be known during development, and it can evolve dynamically over time. For instance, if you start with storing products and only have the name, description, and price, you can easily add more properties later as needed. Additionally, hierarchical structures are simple to form because the values can have subobjects, enabling natural modeling.

In MongoDB, foreign key relationships are rare as the data is often denormalized, resulting in strongly hierarchical objects that are often copies of other places. However, MongoDB now allows schema validation to some extent, as complete schema freedom may not always be optimal. Despite having some level of schema validation, you'll still have a certain base definition, and it's common to define different collections of similar types, making queries more manageable.

---

[1] Initially, MongoDB was released under the GNU APGLv3 license. However, because some Chinese cloud providers were using MongoDB commercially without contributing back to MongoDB Inc., the company behind the database, MongoDB changed its licensing model to require payment from cloud providers. This move away from traditional open-source licensing caused some Linux distributions to stop including MongoDB in their software packages. Unfortunately, this situation isn't unique to MongoDB, as many open-source projects face similar challenges, where significant resources are invested in software development, but the open-source nature of the project means that anyone can profit from it without contributing back to the development team.

"Humongous"

In addition to its schema flexibility, MongoDB also offers the advantage of being able to handle very large databases. This is made possible through *replication* and *sharding*, which allow for high availability and distribution of information across multiple server nodes. MongoDB's ability to handle massive amounts of data has earned it the nickname "humongous"—a play on words with the word "mongo." Some impressive examples of large-scale MongoDB implementations can be found on the MongoDB website (*www.mongodb.com/mongodb-scale*). For instance, the Chinese search engine company Baidu reportedly stores more than 200 billion documents in a MongoDB cluster with 600 nodes, requiring over 1 petabyte (PB) of storage. However, as this data is from 2016, it's likely that even larger implementations have been achieved since then.[2]

When considering the massive scale at which MongoDB can operate, you might wonder if it's suitable for normal users. However, it's important to note that traditional RDBMSs can also handle large amounts of data. For example, Yahoo announced in 2008 that they manage 2 PB of data and perform 24 billion updates per day with PostgreSQL databases. Therefore, the size of the data may not be the main criterion for choosing a database system. Instead, MongoDB's replication mechanism and schema flexibility are significant advantages that allow it to store and handle any type of document in a collection.

8.2.3 Install and Start the MongoDB Community Server

In the following, we want to address a MongoDB server with Spring Boot. Of course, we need a server and a database for this.

The following options are available:

- MongoDB can be run straightforwardly in a Docker container (`docker run -d --name products -p 27017:27017 mongo:latest`).
- MongoDB Inc. provides a *database as a service* (DBaaS) for MongoDB through its project called *MongoDB Atlas* (*www.mongodb.com/atlas*). This cloud-based service allows users to easily deploy, manage, and scale their MongoDB databases without the need for infrastructure management. MongoDB Atlas offers a range of features, including automated backups, monitoring, and security measures such as encryption of data at rest and in transit. The service is available on major cloud providers such as Amazon Web Services (AWS), Microsoft Azure, and Google Cloud Platform (GCP), providing flexibility and accessibility to users. Additionally, MongoDB Atlas offers a free tier plan that allows users to experiment and test their applications with a limited number of resources.
- Download and run MongoDB.

---

[2] *www.mongodb.com/blog/post/mongodb-at-baidu-powering-100-apps-across-600-nodes-at-pb-scale*

Let's stay with the third option. MongoDB can be downloaded for various operating systems at *https://mongodb.com/download-center/community*. Version 6.0 is supported by Spring. Let's download the ZIP archive and unpack it. After that, a script for starting the server can be found in the *bin* directory.

The installation can be tested with the following:

```
$ mongod --version
db version v6.0.0
Build Info: {
    "version": "6.0.0",
    "gitVersion": "e61bf27c2f6a83fed36e5a13c008a32d563babe2",
    "modules": [],
    "allocator": "tcmalloc",
    "environment": {
        "distmod": "windows",
        "distarch": "x86_64",
        "target_arch": "x86_64"
    }
}
```

Starting the MongoDB Community Server

MongoDB requires a directory to store the databases. By default, MongoDB stores the data in the root directory under *data\db*; this directory must exist or an error will occur.

There are two options:

- **Option 1**: The directory is created under *C:\data\db\* (Windows) or *\data\db\* (Unix). The server is started by the program mongod: `start mongod` (Windows) or `mongod` (Unix). Parameters aren't necessary.
- **Option 2**: Another directory is possible, but this must be specified. Let's assume that in the user directory, there is the subdirectory *MongoDB\data*. Then, the server can be started with `start mongod --dbpath=%USERPROFILE%\MongoDB\data` (Windows) or `mongod --dbpath=${HOME}\MongoDB\data` (Unix).

8.2.4 GUI Tools for MongoDB

Different tools are available to access the MongoDB database:

- The installation provides a command-line client with the *mongo shell*.
- Graphical interfaces can be used, for example, the *Compass* application developed by MongoDB Inc. (download is available at *https://mongodb.com/products/compass*).

- Plug-ins are also available for development environments to view and modify data in MongoDB:
  - IntelliJ Ultimate has database tools not only for RDBMS but also for MongoDB.
  - Plug-ins are also available for all other development environments and super editors such as Visual Studio Code (*https://code.visualstudio.com/docs/azure/mongodb*).

8.2.5 Spring Data MongoDB

Spring applications can conveniently interact with MongoDB databases through Spring Data. *Spring Data MongoDB*, a member of the Spring Data family (more information can be found at *http://projects.spring.io/spring-data-mongodb*), offers a collection of utility classes for seamless access and supports the familiar Spring Data features such as repositories, derived query methods, auditing, and more.

When working with MongoDB through Spring Data, it's important to note that the version of Spring Data should match the version of MongoDB to ensure compatibility. The supported versions of Spring Data and MongoDB are listed in the documentation at *https://docs.spring.io/spring-data/mongodb/docs/current/reference/html/#compatibility.matrix*.

Spring Data Release Train	Spring Data MongoDB	Driver Version	Server Version
2023.0	4.1.x	4.9.x	6.0.x
2022.0	4.0.x	4.7.x	6.0.x
2021.2	3.4.x	4.6.x	5.0.x

Table 8.1 Support for MongoDB through Spring Data

Using a compatible version of Spring Data with MongoDB ensures that the application works as expected, without any unexpected behavior or issues. Therefore, it's crucial to keep the compatibility in mind while selecting the versions of Spring Data and MongoDB to use in your project.

Spring Boot Starter MongoDB

There is a starter for Spring Boot. The following must be included in the project object model (POM):

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-mongodb</artifactId>
</dependency>
```

In addition to the dependencies of Spring parts, it integrates the drivers of MongoDB.

When the MongoDB server is running and the Java program is started, there are a number of new messages as the Spring application checks the connection options to MongoDB.

MongoDB Configuration Data

If the MongoDB server is running on your own computer, no further settings are necessary, and the connection can be established directly. However, two things should be added to the configuration properties: the first setting sets the default name of the database to `products`. The next setting causes logging of the data that the Java application exchanges with the MongoDB server.

```
spring.data.mongodb.database=products
logging.level.org.springframework.data.mongodb.core.MongoTemplate=DEBUG
# spring.data.mongodb.host=localhost
# spring.data.mongodb.port=27017
# spring.data.mongodb.uri=\
# mongodb+srv://<username>:<pwd>@<cluster>.mongodb.net/products
```

Listing 8.1 application.properties

Of course, other parameters, such as the server URL or the username, can be configured as usual. You can find some examples commented out. For example, if the server isn't on your own machine, `spring.data.mongodb.host` determines the server, and `spring.data.mongodb.port` assigns the port address. The default port is 27017.

`spring.data.mongodb.uri` can be used to encode all connection data in just one URI. The URI has the format `mongodb+srv://<username>:<pwd>@<cluster>.mongodb.net/db-name`. This is useful for the *Atlas* software; you'll encounter the URI there again.

8.2.6 MongoDB Application Programming Interfaces

Via the *Spring Boot Starter MongoDB*, we get four APIs to access a MongoDB database:

- The `MongoClient` is the native API of the Java MongoDB driver.[3] At the start of the program, the type already appeared briefly in the log. Spring works internally with this type, but it's not of interest to us.
- The first interesting Spring Data MongoDB data type is `MongoTemplate`. The class has similar methods as `MongoClient`.
- The Spring Data MongoDB project provides practical `*Repository` implementations.
- Another option is a special project called *Spring Data MongoDB Reactive* for reactive programming. It provides further support for map-reduce streaming.

We'll look at examples of the API from point 2 and point 3 later.

---

[3] *www.mongodb.com/docs/drivers/java/sync/current/fundamentals/connection/connect*

8.2.7 MongoDB Documents

Jakarta Persistence is primarily geared toward entity beans, whereas MongoDB is focused on documents. The advantage of using MongoDB is that it can conveniently store almost any Java object as a document, providing *schema freedom*. However, this schema freedom can be limiting when trying to retrieve Java objects later on, as Java is a statically typed language and requires a class as a blueprint for the basic structure.

In the following, we want to build a product database for practice purposes. The following class stores the central data of a product.

```
import org.bson.types.ObjectId;
// import org.springframework.data.mongodb.core.mapping.Document;

public class Product {
  public ObjectId id;
  public String title;
  public String description;
  public String thumbnail;
  public String category;
  public long price;  // price in Cent
  // public Map<String, Map<?, ?>> map;
  // public List<Map<String, Map<?, ?>>> listOfMaps;
}
```

Listing 8.2 Product.java

A product has an ID, a title, a description, a thumbnail, and a price (in cents).

Every document must have an ID variable. Either it's really called `id`, or a differently named variable is annotated with `@org.springframework.data.annotation.Id`[4]. An ID field is mapped to `_id` in the MongoDB document. For the ID, the product class uses a special data type `ObjectId` of MongoDB's binary serialization format Binary JSON (BSON).[5] The data type comes from the `org.bson.types` package. A `String` as a data type is possible, but `ObjectId` is the native data type of the MongoDB database. If you reason with the onion architecture, then the technological datatype should not appear in our case—`String` would be acceptable—but for our example, `ObjectId` is fine.

Because a MongoDB document can reference additional fields or child objects, these can be captured in a `Map<String, Map<?, ?>>` or `List<Map<String, Map<?, ?>>>`.

The document class has public instance variables, but setters/getters are also possible; then the variables will remain private. An extremely compact container is created via the public variables. A document class can have a parameterless constructor or a parameterized constructor.

---

[4] *https://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/annotation/Id.html*

[5] *www.mongodb.com/docs/manual/reference/bson-types/*

@Document

While almost any Java object can be stored in MongoDB, a few customizations can be helpful. MongoDB stores documents in different collections, and which collection our class goes into can be determined via `@Document`:[6]

```
@Document( collection = "products" )
public class Product { … }
```

The name of the collection is optional because, by default, it's the name of the class. However, the annotation can be used to explicitly specify the name of the collection.

8.2.8 MongoTemplate Class

The class `MongoTemplate`[7] is very similar in function to `JdbcTemplate`: it's a facade around the native MongoDB Java API. We could build a `MongoTemplate` object manually, but an instance is built automatically via auto-configuration and can be injected:

```
@Autowired
private MongoTemplate mongoTemplate;
```

The `MongoTemplate` class implements the `MongoOperations` interface, which prescribes innumerable CRUD operations. These include `insert(…)`, `findById(…)`, and `findAll(…)`. We'll look at a small subset of the methods. The reference documentation of the Spring Data MongoDB project (*https://docs.spring.io/spring-data/mongodb/docs/current/reference/html/*) and the Javadoc for `MongoTemplate` reveal all the details.

Save a Document

Because our collection is new and consequently empty, we want to store demo data of generated products.

```
for ( int i = 0; i < 100_000; i++ ) {
  var product = new Product();
  product.title = "Product " + i;
  product.description = "Description " + i;
  product.category = "Thing";
  product.price = ThreadLocalRandom.current().nextInt( 1000, 100000 );
  product.thumbnail = "https://picsum.photos/200/300?image=" + i;
  mongoTemplate.save( product );
}
```

Listing 8.3 ProductDatabaseApplication.java, createNewDocuments(), Excerpt

---

[6] *https://docs.spring.io/spring-data/mongodb/docs/current/api/org/springframework/data/mongodb/core/mapping/Document.html*

[7] *https://docs.spring.io/spring-data/mongodb/docs/current/api/org/springframework/data/mongodb/core/MongoTemplate.html*

The `save(…)` method comes from the `MongoTemplate`.

After starting the program, 100,000 products are written to MongoDB. This takes some time because every save is reported due to the log level.

Mapping Annotations

From the example, you can see that saving is simple, and MongoDB does a lot automatically. Of course, mappings can be controlled more precisely via annotations. A selection of annotations is given here:

- `@Document`: This annotation is used to mark a class as a domain object that will be persisted in MongoDB. It is used at the class level and can optionally specify the collection name. If the collection name is not provided, it will be derived from the lowercase class name.
- `@Id`: This annotation is used to mark a field as the primary identifier of the document in MongoDB. Only one field in a class can be marked with `@Id`.
- `@Field`: This annotation is used to specify the name of the field in the MongoDB document that maps to a particular property in the class. It can be used to map a property with a different name in the database.
- `@PersistenceConstructor`: This annotation is used to mark a constructor that Spring Data MongoDB should use when creating instances of the domain object from the database. It is useful when the class has multiple constructors, and you want to specify which one to use for object creation.
- `@Version`: This annotation is used for optimistic locking. It is applied to a field of numeric type (e.g., int, long) and helps to prevent concurrent modifications to a document.
- `@DBRef`: This annotation is used to create a reference to another document in a different collection.
- `@Indexed`, `@CompoundIndex`, `@GeoSpatialIndexed`: These annotations are used to create indexes of fields for improved query performance. `@Indexed` is used for simple indexes, `@CompoundIndex` is used for compound indexes (indexes of multiple fields), and `@GeoSpatialIndexed` is used for geospatial indexes.
- `@TextIndexed`, `@Language`: These annotations are used for full-text search support.
- `@Transient`: This annotation is used to mark a field as transient, meaning it will not be persisted in the database. It is useful when you have fields that you don't want to store permanently.
- `@Value`: Refers to constructors with transformation possibilities, for example, as `@Value("#{args[0] + '.'}")`.

For further annotations, see the documentation at *https://docs.spring.io/spring-data/mongodb/docs/current/reference/html/#mapping-usage-annotations* to get an overview.

Find Everything

The `MongoTemplate` offers not only the possibility to insert data but also to query data. We'll look at two possibilities. The first one is to ask for all the data.

```
System.out.println( mongoTemplate.findAll( Product.class ) );
```

Listing 8.4 ProductDatabaseApplication.java, run(), Excerpt

The `findAll(…)` method is passed a collection type, and it returns all the data.

To query all data is hopeless with a "humongous" database. Therefore, we want to include a query to be able to say exactly which data is searched for.

Search with Query

To query MongoDB, a query can be composed of individual criteria. There are different approaches. One variant works as shown in this listing.

```
Criteria categoryIsThing = Criteria.where( "category" ).is( "Thing" );
Criteria priceRange = Criteria.where( "price" ).gt( 10_00 ).lt( 20_00 );
```

Listing 8.5 ProductDatabaseApplication.java, searchWithCriteria(), Snippet

The `Criteria`[8] objects could be built using the static `where(…)` method or the parameterized constructor. A Fluent API further configures the `Criteria` object.

The criteria can be packaged in a `Query` in the next step.

```
Query query = Query.query( categoryIsThing )
                   .addCriteria( priceRange )
                   .with( Sort.by( "price" ) );
```

Listing 8.6 ProductDatabaseApplication.java, searchWithCriteria(), Snippet

The `Query`[9] object can be passed to various `MongoTemplate` methods, such as `count(…)`, `exists(…)`, or also `find(…)`:

```
List<Product> products = mongoTemplate.find( query, Product.class );
System.out.println( products );
```

The query returns all "things" that cost between 10 and 20, sorted by price.

The query generated by Spring Data MongoDB is displayed via the logger:

```
find using query: { "category": "Thing", "price": {"$gt": 1000, "$lt": 2000}}
```

---

[8] *https://docs.spring.io/spring-data/mongodb/docs/current/api/org/springframework/data/mongodb/core/query/Criteria.html*

[9] *https://docs.spring.io/spring-data/mongodb/docs/current/api/org/springframework/data/mongodb/core/query/Query.html*

8.2.9 MongoDB Repositories

The `MongoTemplate` is one option to access MongoDB. However, Spring Data MongoDB also declares repositories, allowing many more Spring Data options, such as derived query methods, pagination and sorting, and so on.

The Spring Data MongoDB project declares its own repository type `MongoRepository`.[10] The type relationships look familiar from `JpaRepository` (see Figure 8.2).

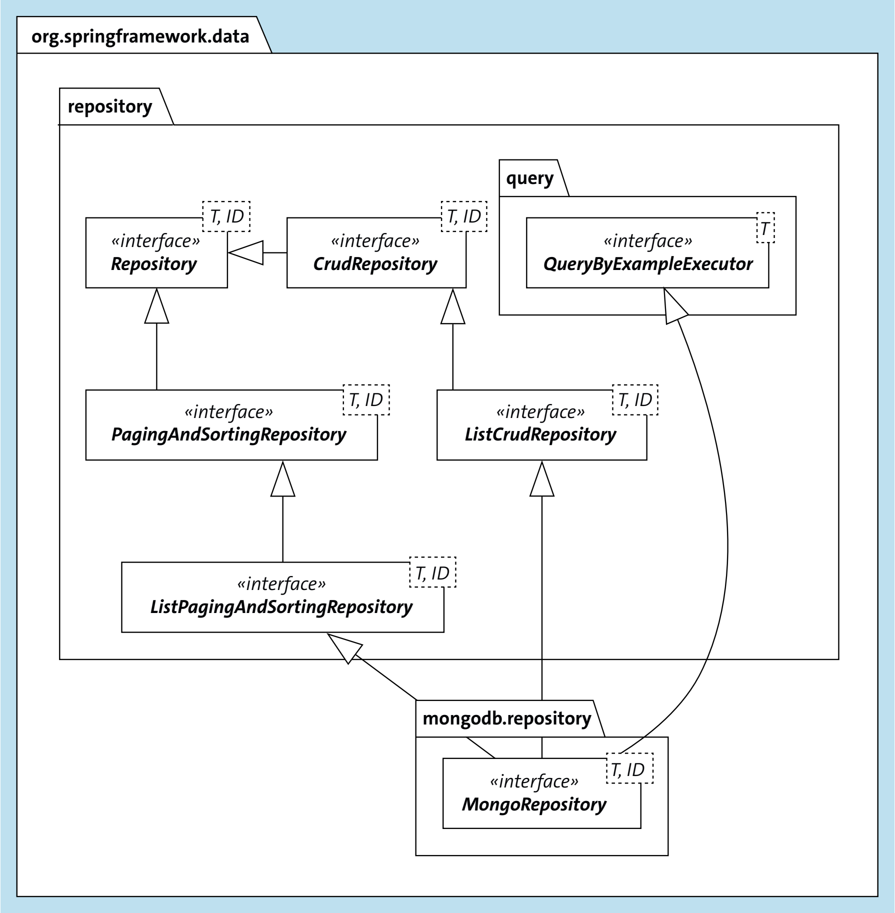

org.springframework.data
repository
query
T, ID
«interface»
Repository
T, ID
«interface»
CrudRepository
T
«interface»
QueryByExampleExecutor
T, ID
«interface»
PagingAndSortingRepository
T, ID
«interface»
ListCrudRepository
T, ID
«interface»
ListPagingAndSortingRepository
mongodb.repository
T, ID
«interface»
MongoRepository

Figure 8.2 "MongoRepository" Type Relationships

The methods of `MongoRepository` are those of the base types, and some methods are overridden with covariant return types. The really new methods are `insert(S entity)` and `insert(Iterable<S> entities)`. While the `save*(…)` methods can update the objects and also create new ones, `insert(…)` may only be applied if the entities are new. MongoDB can do this in a more optimized way than `save*(…)`.

---

[10] *https://docs.spring.io/spring-data/mongodb/docs/current/api/org/springframework/data/mongodb/repository/MongoRepository.html*

Create ProductRepository

At the beginning, as usual, there is a repository interface:

```
interface ProductRepository
extends MongoRepository<Product, ObjectId> { }
```

Our `ProductRepository` gets the two type arguments `Product` and `ObjectId` for the entity type and the key type. In principle, it can also be inherited from `CrudRepository` or from `Repository`.

In the repository, custom methods are possible, which are annotated with `@Query`. The query language isn't Jakarta Persistence Query Language (JPQL) or SQL, but *query documents*.[11] These are nested JSON documents that aren't particularly easy to read:

```
@Query( "{ 'category' : ?0 }" )
List<Product> findProductByCategory( String category );
```

The notation `?0` indicates the first parameter from the method.

Derived query methods are also possible and very welcome. The reference documentation of the Spring Data MongoDB project at *https://docs.spring.io/spring-data/data-mongodb/docs/current/reference/html/#mongodb.repositories.queries* gives some examples. Table 8.2 shows a small selection.

Keyword	Example	Query Document
`After`	`findByBirthdateAfter(Date date)`	`{"birthdate" : {"$gt" : date}}`
`GreaterThan`	`findByAgeGreaterThan(int age)`	`{"age" : {"$gt" : age}}`
`LessThanEqual`	`findByAgeLessThanEqual(int age)`	`{"age" : {"$lte" : age}}`
`Between`	`findByAgeBetween(int from, int to)`	`{"age" : {"$gt" : from, "$lt" : to}}`
`In`	`findByAgeIn(Collection ages)`	`{"age" : {"$in" : [ ages…]}}`
`Containing on String`	`findByFirstnameContaining(String name)`	`{"firstname" : name} (name as regex)`
`Not`	`findByFirstnameNot(String name)`	`{"firstname" : {"$ne" : name}}`

Table 8.2 Example of Derived Query Methods with Generated Query Documents

---

[11] For an introduction, see *www.mongodb.com/docs/manual/tutorial/query-documents/*.

MongoDB offers the possibility of a proximity search—most regular RDBMSs can't do that. The supported keywords are `Near` and `Within`. Another convenient option is regular expression search. In addition, MongoDB also allows full-text searches with a special data type `TextCriteria`. The types `Sort` and `Pageable` can be passed as known.

Inject and Use Repository

Spring implements our `ProductRepository` interface independently, as usual. One instance can be injected:

```
@Autowired
private ProductRepository repository;
```

Now the methods can be called:

```
System.out.println( repository.findAll() );
```

It's already noticeable at this point that, in contrast to the `MongoTemplate`, no type variable is necessary because the type is part of the repository.

8.2.10 Test MongoDB Programs

Applications that use a MongoDB database must of course also be tested. There are two different approaches to testing.

We discussed the Testcontainers project in Section 7.16.6, which can be used to launch databases for a test case in a Docker container. Besides RDBMSs, there are other servers in the Testcontainers project, including MongoDB. That means we would include a dependency on the Testcontainers project and then another dependency for Testcontainers MongoDB (`org.testcontainers:mongodb:x.y.z`), and then we could use the usual mechanisms to program the test case. The production system and the test system are then identical.

Another project, *Embedded MongoDB* (*https://github.com/flapdoodle-oss/de.flapdoodle.embed.mongo*), allows you to run MongoDB outside a Docker container, as a separate process. If you're running a test case for the first time, Embedded MongoDB will download MongoDB from the internet and copy it to a (specified) directory. Once the Spring Boot test is initiated, MongoDB is started as a background process, with preconfigured connections that allow the test to communicate with the MongoDB database. This approach offers a convenient way to run MongoDB for testing purposes without relying on external services or resources.

Documented problems with the library are MongoDB zombies, which are database instances started multiple times that fill up memory and slow down the system.

8.3 Elasticsearch

In this section, we'll look at the Elasticsearch text search engine. We'll first discuss why a special NoSQL database management system like Elasticsearch is necessary and why a normal relational database isn't sufficient.

17th ce
Unicornus
MAGIC

8.3.1 Text Search Is Different

In everyday life, there are always requirements like this:

- Implement a search for articles that makes it possible to search by name, while tolerating typos.
- In the company, we have a large collection of articles, ads, news articles, and PDFs with invoices. Implement a search that finds all documents with the desired keywords.

When dealing with a relational database that only contains text columns, certain problems can't be resolved. While tables are effective at storing well-structured data, text presents a unique challenge. Text searches, in particular, require additional considerations and strategies to be effective:

- Linked searches with logical operators are desirable so that you can say the following, for example: a text should contain “cream” and “candy”, but not “lollipop”.
- The next step is a *fuzzy search*. Text search engines typically reduce words to their root form when storing them. This means that when searching for “drives cars”, documents containing “drive car” will also be recognized, as the text in the database is stored in the root form of “drive” and “car”.
- Word similarities also play an important role; even a phonetic match may be used.
- In many texts, stop words don't play a role in the search and should be ignored. These include “the,” “and,” “a,” and so on. Anything can be a stop word that should be ignored when saving.

- A *proximity search* is helpful. The original document may contain more or fewer words than the search query. This means that the search term “fast car” should also be found if the document contains “fast red car” or vice versa.
- A ranked list of hits would be optimal. It’s also desirable for a text search engine to rank the search results and report that some queries resulted in a 100% match or, as in the last case, came close to that value. The search engine can then report “matches 100%” or “matches 40%.”

RDBMSs can perform text searches with detours, but customization or a supplementary package is always required. For these reasons, special search engines for text queries were developed many years ago.

8.3.2 Apache Lucene

One of the most important products on the market that is also programmed in Java is *Lucene* (*https://lucene.apache.org*) from the Apache Foundation. Lucene is a robust and frequently used Java library for full-text search and is in use in almost every company (*https://cwiki.apache.org/confluence/display/lucene/PoweredBy*).

The open-source library does everything needed for indexing and searching: parsing, analyzing and storing the data, filtering, querying, and caching. Lucene is independent of the document format and stores only the texts. For more complex document formats, such as RTF, Word, or PDF, special extractors are used.

Note

The *Hibernate Search* project (*https://hibernate.org/search/*) extends Hibernate to pass data to Lucene in the background, for example.

8.3.3 Documents and Fields

In Lucene, the focus is on documents, which consist of a collection of fields, such as `nickname` and `description`. The terms are related to MongoDB, which also has documents with fields. Fields themselves consist of key-value pairs. This could look like this:

nickname → “fillmorefat”

description → “coolest unicorn on this planet and beyond”

8.3.4 Index

Lucene performs several operations, collectively referred to as *indexing*, to store text documents. One key operation is *tokenization*, which involves breaking down the document into a sequence of *terms*, usually separated by white space. Case sensitivity is typically ignored during tokenization. Stop words are then removed in the subsequent step, followed by *normalization*. During normalization, plural words can be converted to singular form, and verbs can be reduced to their base form through a process called *stemming*. This process can be configured in various ways, and it's also possible to use and replace synonyms. Overall, Lucene's indexing process is a crucial step in making text documents searchable and easily retrievable.

Let's take an example with a whitespace tokenizer, which is usually used for text. The text "coolest unicorn on this planet and beyond" could be decomposed into the following terms: "cool", "unicorn", "planet", "beyond". Lucene stores these terms exactly, and if you want to express it differently, the terms are indexed tokens.

Inverted Index

After tokenization, an inverted index is built. Each term goes back to the document via this inverted index. Let's look at three documents as an example:

1. Call my **dad**, my mom's too busy
2. Old age isn't so **bad** when you consider the alternatives
3. It's really **bad** when **dads** cry

Some words are in bold because they appear in the various documents.

Text search engines store the terms and to which documents and fields they belong. This is called an *inverted index*. For our three texts, this can look like Table 8.3.

Word	age	alternative	bath	busy	call	consider	cry	dad	mom	old
In the Document	2	2	2, 3	1	1	2	3	1, 3	1	2

Table 8.3 Example of an Inverted Index

If the question later reads, "In which document does 'age' occur?", this database tells us, "It occurs in document 2."

Table 8.3 points out that not all words appear. Stop words have been removed, i.e., words that are undesirable in the index. In our example, these are *it*, *is*, *my*, *not*, *really*, *so*, *the*, *too*, and *when*. It's a matter of configuration what is a stop word. Of course, a word like "not" can also be a term in the index.

8.3.5 Weaknesses of Apache Lucene

Apache Lucene features an indexer that manages documents with an inverted index in the file system. However, Lucene is unable to extract data from specific document formats, such as PDF or HTML. As a result, additional libraries such as *Apache Tika* (*https://tika.apache.org*) are often used to extract text from various document formats. These libraries allow Lucene to access and index the text content of documents in a wide range of formats, making them easily searchable and retrievable.

One of the limitations of Lucene is that it's a traditional Java library you embed and not a server, which means that it can't receive new documents in the index or search for them through a web service call. If a Python program wants to interface with Lucene, for instance, a Java virtual machine (JVM) must always run in the background, and a protocol must be established between the two tools.

Another issue is that Lucene's scalability is limited. The library can only scale vertically, meaning that it requires more computing power to process larger indexes. Because Lucene stores documents in a directory, it's not a typical server that can be scaled horizontally with load balancing. Therefore, there is no failover support.

However, there are solutions to these limitations available in the form of extensions built on top of Lucene. Just like Apache Tika extends Lucene's functionality, these overlays address these weaknesses, enabling Lucene to be scaled up to a much larger size in a cloud environment. By addressing these limitations, these extensions make Lucene a more versatile and scalable tool for indexing and searching text documents.

8.3.6 Lucene Attachments: Elasticsearch and Apache Solr

This is where solutions such as *Elasticsearch* (*www.elastic.co/elasticsearch*) or *Apache Solr* (*https://solr.apache.org*) come into play. Elasticsearch is also a Java software, and it relies on Apache Lucene, as Figure 8.3 shows schematically.

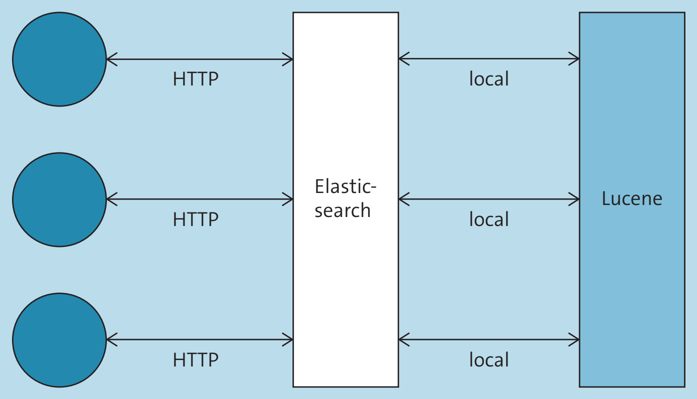


Figure 8.3 Elasticsearch: A Lucene Frontend

While Lucene does the actual work, Elasticsearch accepts requests over HTTP, using local method calls to communicate with the Lucene library.

Along with Elasticsearch, the second solution, Apache Solr, comes from the Apache Foundation itself. However, Elasticsearch as a product is significantly more widespread than Apache Solr.[12]

---

[12] *https://stackshare.io/stackups/elasticsearch-vs-solr* shows 40 times higher penetration in technology stacks.

8.3.7 Install and Launch Elasticsearch

If you're considering using Elasticsearch as your database, there are a few options available to you. First, you can download the Elasticsearch software and run it on your local machine. This option provides you with full control over your database and the flexibility to customize it per your needs.

Second, you can use Elasticsearch in a Docker container. This approach is useful if you want to run Elasticsearch as a part of a larger application or if you want to perform test cases. Docker containers allow you to run Elasticsearch in an isolated environment, without affecting other applications or systems running on your machine.

Lastly, you can also use a hosted version of Elasticsearch in the cloud. Many providers offer Elasticsearch hosting services, including the company behind Elasticsearch, Amazon, and Azure. Hosted Elasticsearch solutions provide you with a fully managed and scalable database service, taking away the hassle of managing infrastructure and enabling you to focus on your application and data. However, this option may come with a cost, and you may have to rely on the provider's infrastructure and support.

Download Elasticsearch

Let's use a local installation, which is easy to use thanks to the implementation in Java. We select the widely used latest version 7 at *www.elastic.co/guide/en/elasticsearch/reference/master/important-settings.html#path-settings* and download it for the operating system. There are several archive formats, including the straightforward ZIP, with which the download would be *elasticsearch-7.x.y-windows-x86_64.zip* on Windows.

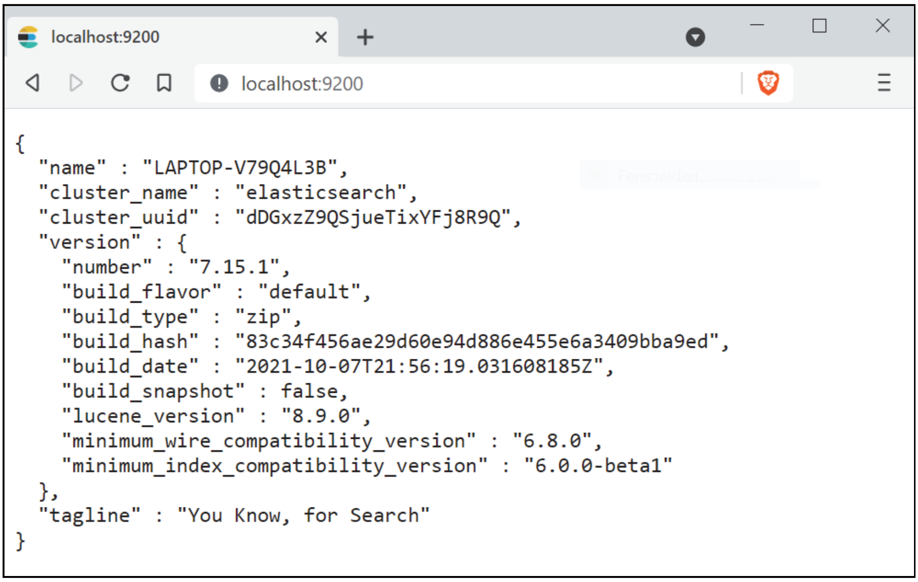
Once downloaded, Elasticsearch needs to be unpacked before use. It's worth noting that the download size is relatively large because Elasticsearch comes bundled with *Eclipse Adoptium* version 17. This approach is practical as it eliminates the need to install Java separately on the computer and increases the likelihood of Elasticsearch working seamlessly without requiring additional configuration. By including the necessary Java components within the Elasticsearch package, users can get started with Elasticsearch more quickly and easily.

In principle, you can start Elasticsearch right away. Parameters are set in the following two ways:

- Via the command line using the `-E` switch.
- Via the configuration file *config/elasticsearch.yml*. You can set data and log directories among other names for the cluster (the default is *elasticsearch*).

For all settings, see *www.elastic.co/guide/en/elasticsearch/reference/7.17/settings.html*. We won't need to make any settings for our example.

In the *bin* directory, there is a batch file called *elasticsearch.bat* in short. With that file, we start the server, which takes some time. If you start a web browser and select *http://localhost:9200*, you should see a message similar to the one in Figure 8.4.



```
{
  "name" : "LAPTOP-V79Q4L3B",
  "cluster_name" : "elasticsearch",
  "cluster_uuid" : "dDGxzZ9QSjueTixYFj8R9Q",
  "version" : {
    "number" : "7.15.1",
    "build_flavor" : "default",
    "build_type" : "zip",
    "build_hash" : "83c34f456ae29d60e94d886e455e6a3409bba9ed",
    "build_date" : "2021-10-07T21:56:19.031608185Z",
    "build_snapshot" : false,
    "lucene_version" : "8.9.0",
    "minimum_wire_compatibility_version" : "6.8.0",
    "minimum_index_compatibility_version" : "6.0.0-beta1"
  },
  "tagline" : "You Know, for Search"
}
```

Figure 8.4 The Web Page at "http://localhost:9200" Showing Information about the Servers

Elasticsearch is running and ready to accept queries. That's why we can switch to the Java side.

8.3.8 Spring Data Elasticsearch

Spring Data Elasticsearch (*https://spring.io/projects/spring-data-elasticsearch*) is a member of the Spring Data family and offers the usual features of Spring Data: repositories, sorting and pagination, derived query methods, projections, and so on.

It's worth noting that Spring Data doesn't support all versions of Elasticsearch. The supported versions are listed in the documentation at *https://docs.spring.io/spring-data/elasticsearch/docs/current/reference/html/#preface.versions*. Therefore, it's essential to check the compatibility between your Elasticsearch version and the version of Spring Data you plan to use.

Spring Data Release Train	Spring Data Elasticsearch	Elasticsearch	Spring Framework	Spring Boot
2023.0	5.1.x	8.7.1	6.0.x	3.1.x
2022.0	5.0.x	8.5.3	6.0.x	3.0.x
2021.2	4.4.x	7.17.3	5.3.x	2.7.x

Table 8.4 Support for Elasticsearch through Spring Data

Spring Boot Starter Data for Elasticsearch

For Elasticsearch, there is a Spring Boot Starter, which can also be selected directly during Initializr. In the POM file comes for the Spring Boot Starter Data Elasticsearch:

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-elasticsearch</artifactId>
</dependency>
```

Settings

As usual, customizations can be made via configuration properties. If Elasticsearch is running on your machine, no customizations are needed because Spring Data Elasticsearch auto-configuration prepares everything. By default, Spring assumes that the server is running on the default port on `localhost`, and the cluster is named `elasticsearch`. It's recommended to set the log level for better traceability at the beginning. Then, Elasticsearch shows what data is exchanged with the server:

```
logging.level.tracer=TRACE
```

Listing 8.7 application.properties Extension

We had already set the log level for other members of the Spring Data family, for example, to see what SQL Hibernate generates or what JSON documents in MongoDB generates.

8.3.9 Documents

As mentioned earlier, Elasticsearch stores documents, and each document consists of key-value pairs called fields. Each field has a type, and besides text, there are different types. These include (as a Java type) `Text`, `Keyword`, `Long`, `Integer`, `Short`, `Byte`, `Double`, `Float`, `Half_Float`, `Scaled_Float`, `Date`, `Date_Nanos`, `Boolean`, `Binary`, `Integer_Range`, `Float_Range`, `Long_Range`, `Double_Range`, `Date_Range`, `Ip_Range`, `Object`, `Nested`, `Ip`, `TokenCount`, `Percolator`, `Flattened`, and `Search_As_You_Type`. Elasticsearch not only stores text but also various data types; therefore, Elasticsearch and MongoDB aren't that far apart.

The Spring Data Elasticsearch project provides several annotation types that developers can use in their classes, such as the `@Document`[13] annotation. These annotations allow for precise description of data types for fields. When fields aren't specified with a data type, Elasticsearch will attempt to determine the type on its own. However, it's better to explicitly specify the data types using annotations whenever possible to avoid any guesswork on Elasticsearch's part. By specifying the types, developers can ensure that

---

[13] *https://docs.spring.io/spring-data/elasticsearch/docs/current/api/org/springframework/data/elasticsearch/annotations/Document.html*

Elasticsearch accurately stores and retrieves data, leading to improved search and query performance.

@Document Message

Let's start with a document that can store messages of a chat history. This document is to be included in an index with the name `messages`. An index is similar to a table in relational databases and to a collection in MongoDB.

```
import org.springframework.data.annotation.Id;
import org.springframework.data.elasticsearch.annotations.*;
import java.time.OffsetDateTime;
import java.util.List;

@Document( indexName = "messages" )
public class Message {

  @Id
  public String id;

  @Field( type = FieldType.Keyword )
  public String conversationId;

  @Field( type = FieldType.Long )
  public List<Long> participantIds;

  @Field( type = FieldType.Text )
  public String message;

  @Field( type = FieldType.Date, format = DateFormat.date_time )
  public OffsetDateTime sent;
}
```

Listing 8.8 Message.java

Let's go through the instance variables:

- Each document is uniquely identified by an ID. The ID field is annotated with `@Id`,[14] and the annotation type comes from *Spring Data Commons*.
- The `conversationId` is a string and is supposed to be composed of the parties chatting with each other. For example, if 1 is talking to 2 (the IDs of the unicorn or profile), the `conversationId` can be the string `1+2`. This allows us to ask for the messages from two chatting unicorns.

---

[14] *https://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/annotation/Id.html*

- In addition, there is a list of longs, `participantIds`, to map the order of the participants. That is, for example, if 1 talks to 2, the list is `[1, 2]`; conversely, if 2 talks to 1, the list is `[2, 1]`. While the `conversationId` is sorted and the smaller ID is in front, we get an order via the list, so we can see who is talking to whom.
- The actual message is in `message`. The field type is `FieldType.Text`,[15] and this is the message we can search for later.
- The `sent` variable contains the time at which the message was sent.

We've parameterized only a tiny part of the annotation `@Field`. If you take a look at the Javadoc,[16] you'll find perhaps the most comprehensive annotation type in the entire Java universe.

Constructors and a `toString(…)` method complete the `Message` class.

```
public Message() { }

public Message( String conversation, List<Long> participants,
                String message, OffsetDateTime sent ) {
  this.conversationId = conversation;
  this.participantIds = participants;
  this.message = message;
  this.sent = sent;
}

@Override public String toString() {
  return String.format( "Message{id='%s', conversationId='%s'," +
                        "participantIds=%s, message='%s', sent=%s}",
                        id, conversationId, participantIds, message, sent );
}
```

Listing 8.9 Message.java Extension

8.3.10 ElasticsearchRepository

Elasticsearch declares its own repository, just like the other Spring Data projects: `ElasticsearchRepository`.[17] The type relationships look familiar, as shown in the Unified Modeling Language (UML) diagram in Figure 8.5.

---

[15] *https://docs.spring.io/spring-data/elasticsearch/docs/current/api/org/springframework/data/elasticsearch/annotations/FieldType.html*

[16] *https://docs.spring.io/spring-data/elasticsearch/docs/current/api/org/springframework/data/elasticsearch/annotations/Field.html*

[17] *https://docs.spring.io/spring-data/elasticsearch/docs/current/api/org/springframework/data/elasticsearch/repository/ElasticsearchRepository.html*

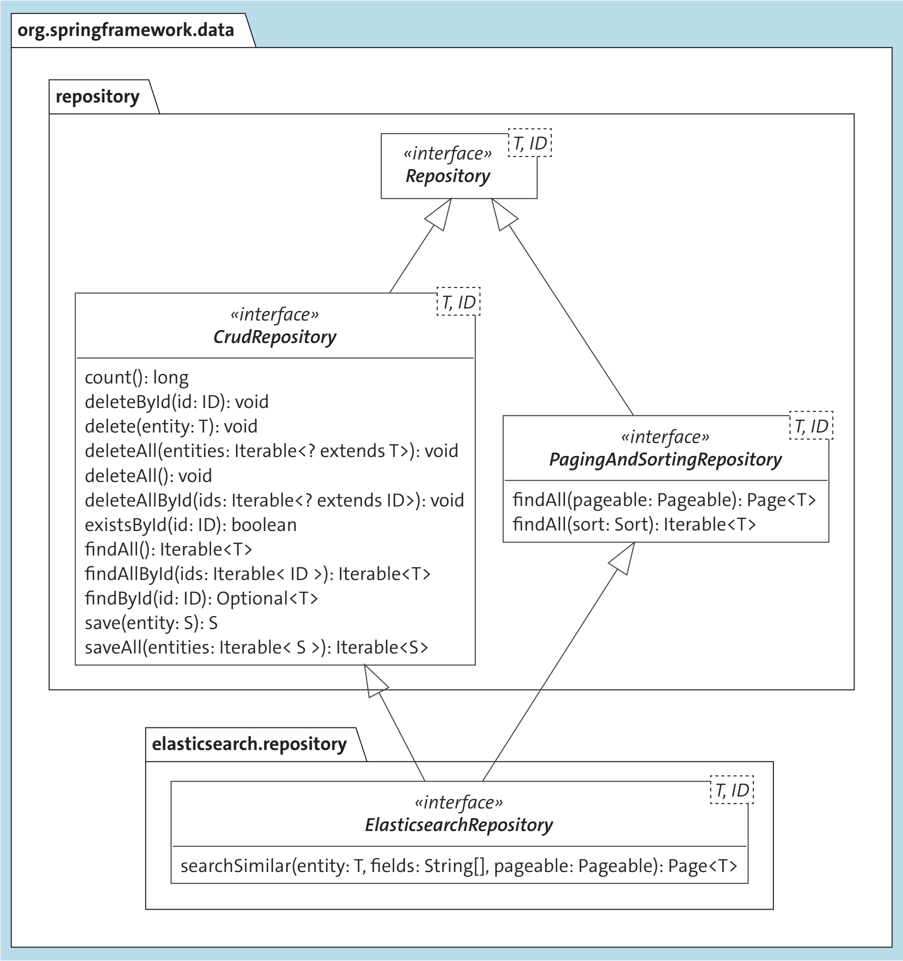

org.springframework.data
repository
«interface»
Repository
T, ID
«interface»
CrudRepository
T, ID
count(): long
deleteById(id: ID): void
delete(entity: T): void
deleteAll(entities: Iterable<? extends T>): void
deleteAll(): void
deleteAllById(ids: Iterable<? extends ID>): void
existsById(id: ID): boolean
findAll(): Iterable<T>
findAllById(ids: Iterable< ID >): Iterable<T>
findById(id: ID): Optional<T>
save(entity: S): S
saveAll(entities: Iterable< S >): Iterable<S>
«interface»
PagingAndSortingRepository
T, ID
findAll(pageable: Pageable): Page<T>
findAll(sort: Sort): Iterable<T>
elasticsearch.repository
«interface»
ElasticsearchRepository
T, ID
searchSimilar(entity: T, fields: String[], pageable: Pageable): Page<T>

Figure 8.5 Type Relationships of “ElasticsearchRepository”

Compared to the well-known `JpaRepository` and `MongoRepository`, there are differences: `ElasticsearchRepository` doesn’t inherit from `QueryByExampleExecutor` and doesn’t overwrite the methods with covariant return types, and the `find*(…)` methods still return `Iterable` instead of `List`. However, we can compensate for these differences by adding derived query methods that have similar formulations and provide better return types. A new method is added: `searchSimilar(…)`. Unfortunately, this method isn’t so useful in practice because its results don’t show how much the results really resemble the original.

Writing Your Own Elasticsearch Repository

`ElasticsearchRepository` can be used in the same way as `JpaRepository` and `MongoRepository`. Here’s an example of how we can define a new interface `MessageRepository` to read and write our `Message` documents.

```
public interface MessageRepository
    extends ElasticsearchRepository<Message, String> {

  List<Message> findMessageByConversationIdOrderBySent(
      String conversationId
  );
}
```

Listing 8.10 MessageRepository.java

With the base type, all CRUD operations are available. Additionally, a derived query method `findMessageByConversationIdOrderBySent(…)` has been added. Elasticsearch has a native query language,[18] but derived query methods are very handy as they can eliminate the need for the long `@Query`. The method is parameterized with a `conversationId` so that we get a list of `Message` objects by the filter criterion.

Spring Data Elasticsearch uses the familiar keywords in derived query methods, but in addition to `List<T>` and `Stream<T>`, there are some interesting new return types[19]: `SearchHits<T>`, `List<SearchHit<T>>`, `Stream<SearchHit<T>>`, and `SearchPage<T>`. With `SearchHits`, not only the search result is returned as a string but also the quality of the search results is communicated.

Using Your Own Elasticsearch Repository

With the document and `MessageRepository` completed, we can look at a full example of how we can store and query `Message` objects in Elasticsearch:

```
@SpringBootApplication
public class SpringBootESApplication1 {
  public SpringBootESApplication1( MessageRepository messages ) {
    messages.deleteAll();
    List<Message> newMessages = Arrays.asList(
     new Message( "1+2", Arrays.asList( 1L, 2L ),
                  "Can I tell you a time travel joke?",
                  OffsetDateTime.now().minusSeconds( 20 ) ),
     new Message( "1+2", Arrays.asList( 2L, 1L ),
                  "Yes, sure",OffsetDateTime.now().minusSeconds(10)),
     new Message( "1+2", Arrays.asList( 1L, 2L ),
                  "You didn't like the joke.",OffsetDateTime.now()),
     new Message( "2+3", Arrays.asList( 2L, 3L ),
                  "I know a joke.", OffsetDateTime.now())
```

---

[18] As with MongoDB, the query is JSON-based; details are available at *www.elastic.co/guide/en/elasticsearch/reference/current/query-dsl.html.*

[19] *https://docs.spring.io/spring-data/elasticsearch/docs/current/reference/html/#elasticsearch.query-methods.criterions*

```
    );
    messages.saveAll( newMessages );
    messages.findAll().forEach( System.out::println );
    messages.findMessageByConversationIdOrderBySent( "1+2" )
            .forEach( System.out::println );
  }
  public static void main( String[] args ) {
    SpringApplication.run( SpringBootESApplication1.class, args );
  }
}
```

Via constructor injection, the program requests a `MessageRepository` and deletes all messages from the database with `deleteAll()` so that the output remains repeatable even with multiple calls.

Then, a list of `Message` objects is built. The first argument in the constructor is the `conversationId`. Here, the IDs of the unicorns communicating with each other are concatenated with `+`. By definition, we make sure that the smaller key is always in front. Because that doesn't tell us anything about the order in which the messages arrived, a list holds that. This is followed by the content of the message and a timestamp. A call to `saveAll(…)` saves the messages.

In the third step, the messages are to be loaded. For this purpose, `findAll()` returns all messages. But that can be too many, and then `findMessageByConversationIdOrderBySent("1+2")` comes into play. We pass the built message ID and get all the messages where unicorn 1 talked to unicorn 2, sorted by date.

8.3.11 @DataElasticsearchTest

Spring Boot also offers various test slices for Elasticsearch. With `@DataElasticsearchTest`,[20] only those components are initialized that have to do with Elasticsearch. For the server, you can again use Testcontainers. The approach is the same as for PostgreSQL in Chapter 7, Section 7.16.6:

```
@SpringBootTest
@Testcontainers
class ESTests {

  @Container
  @ServiceConnection
  static ElasticsearchContainer testContainer =
    new ElasticsearchContainer(
       "docker.elastic.co/elasticsearch/elasticsearch:7.17.12"
```

---

[20] *https://docs.spring.io/spring-boot/docs/current/api/org/springframework/boot/test/autoconfigure/data/elasticsearch/DataElasticsearchTest.html*

```
    ).withEnv( "discovery.type", "single-node" );

  @Test …
}
```

In the test case, the Docker container is started, and the properties are set.

8.4 Summary

At the end of this chapter, it's important to note that while there is only one relational model, there are countless nonrelational options, which is why the Spring Data family is so extensive. Spring Data offers implementations for a variety of NoSQL databases and storage systems, with *Spring Data Cassandra*, *Spring Data Couchbase*, and *Spring Data Neo4j* being particularly relevant in practice. Another important implementation is *Spring Data Redis*, which is commonly used as a distributed cache.

For those looking to learn more about Redis, a good place to start is by implementing a Twitter clone with Redis, which can be found at *https://redis.io/docs/reference/patterns/twitter-clone/*. The implementation can be done using Spring technologies, with an example available at *https://docs.spring.io/spring-data/data-keyvalue/examples/retwisj/current*. Although the example is 10 years old, it remains relevant today.

It's worth noting that the aforementioned implementations are official solutions from VMware. However, there are other open-source implementations available as well, but their quality and support for Spring Data may vary and must be closely monitored.

Chapter 9

Spring Web

In this chapter, we'll delve into the world of the HTTP web protocol, web pages, and dynamically generated documents. However, before we jump into the specifics of how Spring initiates a web server and serves static and dynamic content, let's take a trip back in time, several decades ago.

Appreciation of Tim Berners-Lee

Tim Berners-Lee, a British physicist and computer scientist, spent a considerable amount of time working at CERN, a prominent European organization dedicated to nuclear research. Berners-Lee and his colleagues encountered a problem when attempting to exchange documents between different computer systems from various countries. As a solution, he and his Belgian colleague Robert Cailliau developed a set of protocols that facilitated document exchange. During this process, they established crucial standards that persist today, including the transfer protocol HTTP (*Hypertext Transfer Protocol*) and the page description language HTML (*Hypertext Markup Language*). The initial web browser and server, the CERN httpd, were products of Tim Berners-Lee's work. (Interestingly, the CERN *httpd* was later replaced by *Jigsaw*, implemented in Java.)

9.1 Web Server

The primary objective of a web server is to provide documents to clients, using either the HTTP transmission protocol or its secure variant, HTTPS. Although there are several web server implementations available, statistics indicate that only two products, *nginx* and the *Apache HTTP Server* (as shown in Figure 9.1), are relevant. These two prominent web servers, along with the LiteSpeed Web Server, are written in C++. Conversely, although there are Java-programmed web servers, they aren't included in the list due to their insignificant distribution.

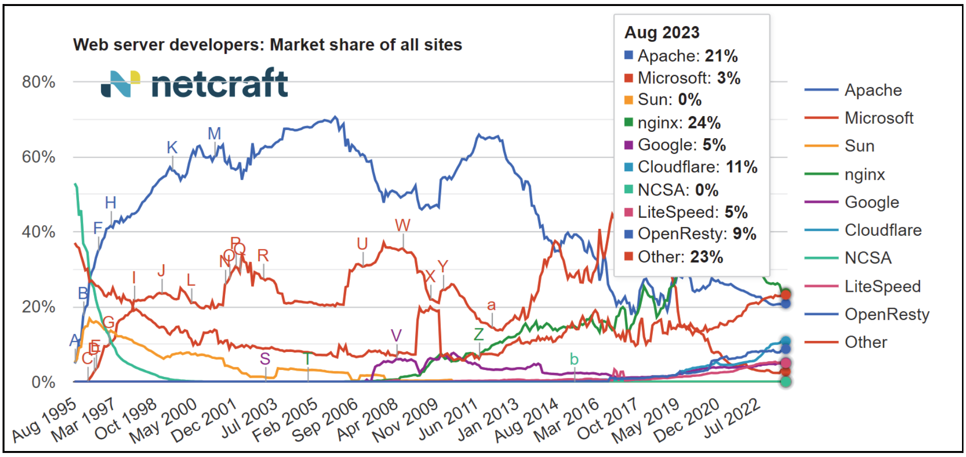


Figure 9.1 Proliferation of Web Servers over the Past Decades (Source: https://news.netcraft.com/archives/category/web-server-survey/)

9.1.1 Java Web Server

There are several web servers implemented in Java, and among the most popular are the following:

- *Apache Tomcat* (*https://tomcat.apache.org*)
- *Jetty* (*www.eclipse.org/jetty*)
- *Undertow* (*https://undertow.io*)

In addition to delivering static documents such as web pages, CSS documents, and the like, a web server can also generate documents dynamically during runtime. In such cases, it's advantageous for the programs responsible for generating dynamic content to be developed in Java and reside within the Java web server, thereby ensuring efficient processing. However, Java-programmed web servers don't have a significant worldwide share. This is primarily because they are superseded by a "genuine" web server, such as nginx, which accepts HTTP requests and differentiates between static and dynamically generated resources during runtime. The main web server itself delivers static resources, while requests for dynamically generated documents are forwarded to a Java web server.

Standalone versus Embedded

Java-programmed servers can be integrated into personal programs, allowing them to become a component of your own software rather than operating independently. This is referred to as an *embedded web server*. This concept was introduced in Chapter 5, Section 5.1.1, where the embedded database management system or embedded web server starts up along with the application. Upon shutting down the program, the embedded servers also shut down. By integrating servers into their own application, users aren't reliant on external solutions, and the servers are embedded as a library. This affords more control over version numbers, allowing for optimal operation between the server and the application. As a result, embedded servers are less dependent on infrastructure services.

9.1.2 Spring Boot Starter Web

A web server can easily be embedded in a Spring application. There is a separate starter for Spring Boot, called the *Spring Boot Starter Web*, which includes an embedded Tomcat server. In the project object model (POM) file, it's added as follows:

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

Listing 9.1 pom.xml Extension

The Spring Boot Starter Web references two important open-source libraries: *Tomcat* and *Jackson*, the latter for a JavaScript Object Notation (JSON) mapping.

The Spring Boot Starter Web also references the *Spring Boot Starter*. So, when we build pure web applications with Spring Boot, we don't need to reference the *core starter*.

Spring Boot Starter Web also references *Spring Web* and *Spring Web model-view-controller* (Spring Web MVC). Spring Web brings core functionality, but a much larger building block is provided with Spring Web MVC. MVC is a widely used design pattern that is commonly divided into three components. The *model* represents the data of the application and handles its storage, retrieval, and management. The *controller* manages the application's logic, directing the flow of data and managing communication between the model and the *view* for the presentation.

We'll briefly look at a part from Spring Web and later work more intensively with Spring Web MVC, especially for implementing RESTful web services.

Log Messages from the Embedded Server

If Spring Boot finds an embedded web server in the classpath, it starts the web server automatically. This can be seen in the log:

```
… o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat initialized with port(s): ↩
                                             8080 (http)
… o.apache.catalina.core.StandardService   : Starting service [Tomcat]
… o.apache.catalina.core.StandardEngine    : Starting Servlet engine:  ↩
                                             [Apache Tomcat/10.1.12]
```

```
… o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring embedded  ↩
                                             WebApplicationContext
… w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: ↩
                                             initialization completed in 683 ms
… o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat started on port(s): ↩
                                             8080 (http) with context path ''
```

Our application boots up, and the usual log messages follow. What's new is the output stating that Tomcat was started on port 8080. The program isn't terminated after startup, and Tomcat waits for incoming connections.

Task: Because the Spring Shell is no longer needed in the following, we can disable it. Put *application.properties* in the following line:

```
spring.shell.interactive.enabled=false
```

9.1.3 Use Other Web Servers *

Tomcat is a well-known product, and therefore the Spring team also uses Tomcat by default. In principle, however, other Java web servers such as Jetty and Undertow, mentioned earlier, can be used. Undertow shows good performance and has a slightly lower memory consumption than Tomcat according to current measurements.

If you want to use other Java web servers, you need to do two things: first remove Tomcat from the POM, and, second, reference the desired web server with a new dependency. The reference documentation at *https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-use-another-web-server* explains the steps in more detail. We want to stay with Tomcat because the server container is well known and excellently documented, and, furthermore, the native compilation supports Tomcat exclusively as a web server so far.[1]

9.1.4 Adjust Port via server.port

Web servers can have many configurations. The most obvious is to adjust the port on which the server listens. By default, a web server is started on port 8080. This is fine for testing. If a web server is also supposed to deliver static web pages on the Internet, port 80 is common.

The port can be changed via the `server.port` configuration property. There are different ways to set it, for example, via *application.properties*, in which it's called `server.port=8888`, or via the command line, in which `-DSERVER_PORT=8888` is passed at startup.

If the web server is to occupy any free port, then set the port to `0`. This is often used for tests, so that parallel tests are also possible.

---

[1] *https://docs.spring.io/spring-native/docs/current/reference/htmlsingle/#_starters_requiring_special_build_configuration*

For further configuration properties, take a look at the reference documentation:

- *https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-customizing-embedded-containers*
- *https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#appendix.application-properties.server*
- *https://docs.spring.io/spring-boot/docs/current/api/org/springframework/boot/autoconfigure/web/ServerProperties.html*

9.1.5 Serve Static Resources

A web server has the capability to perform two primary functions: serving static resources and generating dynamic content at runtime. Static resources typically include items such as CSS documents, JavaScript programs, or images that don't require runtime processing.

In the case of Spring, the framework searches specific directories for resources that are accessed on the server. For instance, when a resource is accessed at *localhost:8080*, Spring Boot automatically searches for the resource within four default directories:

- */static*
- */public*
- */resources*
- */META-INF/resources*

If you create a project via the Initializr and check the box for **Spring Web**, then the *static* directory is automatically created. Because we started with the core launcher in the project, the directory is missing. Therefore, for an example, the Maven default directory layout should be created under *src/main/resources/static* in the directory structure.

If we place an HTML file such as *index.html* under the directory, the web page can be seen later at *http://localhost:8080/index.html*. Calling *http://localhost:8080/* automatically redirects to *index.html*.

Note

Reloading a modified web page in the browser isn't an effective solution because Spring caches the file contents on the server side. This means that any changes made to the page won't be reflected until the cache is cleared. When a live server is running, which we'll see in a minute, the cache is automatically cleared after a build, allowing for the updated content to be served to users.

9.1.6 WebJars

Larger Java programs always have dependencies. The Maven Central server is our biggest friend because it provides a gigantic number of Java archives. It's no different with web development. There, too, dependencies exist on web frameworks, on CSS files, and so on. However, the way dependencies are handled is quite different. Therefore, it's convenient that there are *WebJars* that package typical web artifacts such as JavaScript programs, CSS files, and so on in a Java archive (JAR). These WebJars, like other Java libraries, are hosted on the Maven Central server, making it easy to include them as a normal dependency in a Java project. This convenience simplifies the process of managing web dependencies and helps developers to focus on building the functionality of their application.

There are numerous WebJars, evidenced at *www.webjars.org* where about 20,000 artifacts are listed, including classics such as *Bootstrap*, *Font Awesome*, and *Chart.js*.

For example, if the popular CSS framework Bootstrap is to be included, the following is placed in the POM:

```
<dependency>
  <groupId>org.webjars.npm</groupId>
  <artifactId>bootstrap</artifactId>
  <version>5.3.1</version>
</dependency>
```

Directory *META-INF/resources/webjars/bootstrap/5.3.1/* is in the JAR. Because we just discussed in the previous section that everything can be retrieved from *META-INF/resources/* in the root directory of the web application, the CSS file can be obtained this way with a reference in the HTML header:

```
<link rel="stylesheet"
      href="webjars/bootstrap/5.3.1/dist/css/bootstrap.css">
```

9.1.7 Transport Layer Security Encryption

In today's internet landscape, encrypting internet connections has become a necessity. Java web servers have the capability to provide encryption through the standard *Transport Layer Security* (TLS), which is the successor of *Secure Sockets Layer* (SSL). TLS and SSL ensure that the message being transmitted over a network is encrypted. For encryption, a digital certificate is required, which is commonly referred to as an SSL certificate. This certificate includes a cryptographic key and metadata.

A *digital certificate* is required for the encryption. which is often also called an *SSL certificate*. This certificate is a small file containing a cryptographic key and metadata.

Digital certificates can be obtained in two ways:

- From a *certificate authority* (CA), that is, an official certification authority
- Through a *self-signed certificate*

Official certificates, also known as CA-signed certificates, are issued by trusted certification authorities and can be used to secure websites over the internet. However, obtaining such certificates can be expensive, especially for small-scale or personal projects.

On the other hand, self-signed certificates can be generated for free and used to secure communication between servers and clients. Self-signed certificates aren't issued by a trusted authority and hence aren't automatically trusted by web browsers. This means that when a user visits a website with a self-signed certificate, their browser will display a warning that the connection isn't secure.

Despite this warning, self-signed certificates can still be useful for development and testing purposes, especially on `localhost` where the website isn't exposed to the public internet.

Create a Digital Certificate with Keytool

You can create your own certificates, for example, with tools from the OpenSSL project and with the JDK `keytool`.[2] The `keytool` can be called from the command line as follows:

```
$ keytool ↩
  -genkeypair ↩
   -alias tutego ↩
   -keyalg RSA ↩
   -keysize 2048 ↩
   -storetype PKCS12 ↩
   -keystore tutego.date4u.p12 ↩
   -validity 365 ↩
   -storepass DOOFDOOF
```

The specifications have the following meaning: `alias` defines the name of the key pair to be generated, `keyalg` defines the algorithm (here, RSA), `keysize` defines the number of bits 2,048 (i.e., 256 bytes), and `storetype` defines the archive format (PKCS #12 is a standardized file format). The file will later have a name; it's called `tutego.date4u.p12`. The validity is set to 365 days with `validity`. The password of the keystore file is `DOOFDOOF`.

A few questions about the name and more follow, and we can enter anything we like at this point. At the end, a certificate file is created, which is about 10 kibibytes (KiB) in size.

Enable HTTPS in Spring Boot

To enable TLS encryption in Spring Boot, we need to announce the digital certificate. To do this, we enter something like the following in the *application.properties* file:

```
server.port=8443
server.ssl.key-store=classpath:tutego.date4u.p12
server.ssl.key-store-password=DOOFDOOF
```

---

[2] *https://docs.oracle.com/en/java/javase/17/docs/specs/man/keytool.html*

The certificate file needs to be in a specific location to be used by the application. Per the specification, the file should be placed under the *src/main/resources* directory. The default port for encrypted connections is 443, which is used in production environments. However, for development and testing purposes, port 8443 is usually used.

Additional settings are shown at *https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto.webserver.configure-ssl*.

9.2 Generate Dynamic Content

Until now, we've discussed how web servers can deliver static content. However, generating dynamic content at runtime is a more exciting and useful feature.

9.2.1 Stone Age: The Common Gateway Interface

In the early days of web development, before the creation of Java or JavaScript, the only way to incorporate dynamic content was through the web server calling an external program that generated web content.

The diagram in Figure 9.2 shows the steps that still apply in essence to modern programs.

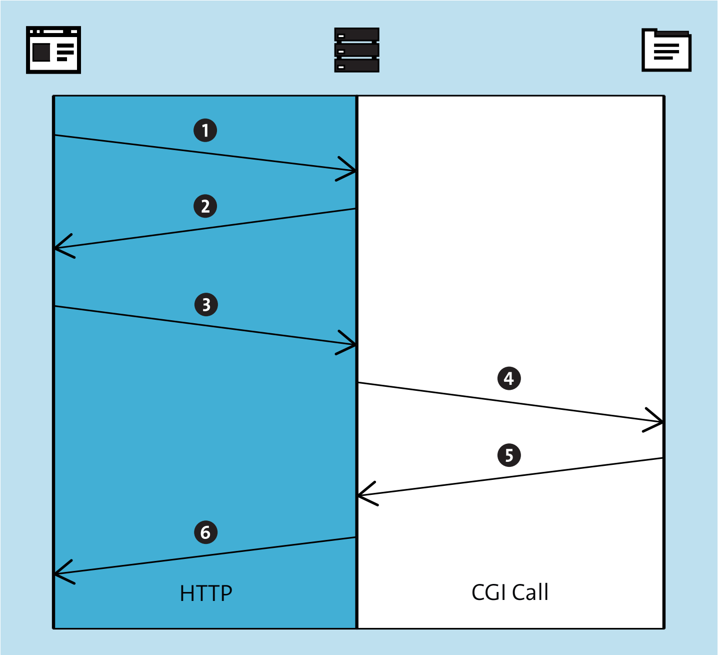


Figure 9.2 Dynamic Content via the Common Gateway Interface (CGI) Standard

The browser starts a request ❶, and the server returns, for example, an HTML form ❷. The browser displays the form, and the user enters and submits the data ❸. The server recognizes that the selected address doesn't represent a static resource, so the server must address an external program ❹ via the *Common Gateway Interface* (CGI) with the external program. The external program generates a response ❺, and the server receives it and returns it to the client ❻.

CGI had a major drawback of increased runtime because the web server had to initiate external processes. This was mainly because web servers couldn't contain user-defined program parts and had to call external programs written in other programming languages such as C or Perl. This led to the server having to start new processes for C programs or constantly restart the Perl interpreter to interpret the programs. However, to speed up communication with the external programs, they were included in the web server itself. This was the origin of servlet technology for Java web servers.

9.2.2 Servlet Standard

In the Java environment, there is the *servlet specification*, the first version of which dates back to late 1996. A servlet is a small Java program inside a *servlet container*. The servlet container, in turn, is a component of a web server programmed in Java (see Figure 9.3).

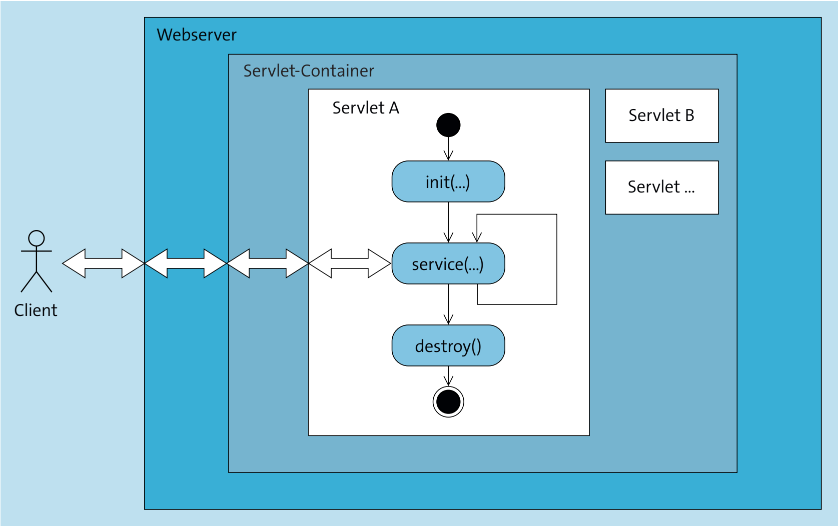


Figure 9.3 Data Flow in a Servlet

When a client sends a request to a web server, the server first checks if the requested resource is a static file or not. If it's a static resource such as an image or a CSS file, the server can directly send it back to the client. However, if the requested resource is a special endpoint that requires dynamic content, the server recognizes this and passes the request to a servlet for further processing. The servlet can then generate the necessary content and return it to the web server, which then sends it back to the client.

The lifecycle of a servlet consists of three stages. The first stage is the initialization stage, where the servlet is started and initialized using the `init(…)` method. This stage happens only once during the entire lifetime of the servlet. The second stage is the processing stage, where the servlet is capable of handling multiple requests, and these requests are received by the servlet in the `service(…)` method.

The servlet can be used as often as desired and isn't always instantiated again. Therefore, the object should also be stateless. Typical web servers in Java are *multithreaded*, which means multiple HTTP clients contact the same servlet, and then the `service(…)` method is also called at the same time because one thread is classically used per incoming connection. The consequence of a hundred incoming connections would also be a hundred threads in the `service(…)` method.

If the servlet is no longer needed later, that is, the web server shuts down with the servlet container, the `destroy()` method should be called by the servlet container. Then the life of the servlet ends.

9.2.3 Program @WebServlet

The presence of several servlets is typical in a servlet container. Our focus, however, is on learning how to code a servlet.

In the realm of servlets, a fundamental building block is the `jakarta.servlet.Servlet`[3] base type that comes as part of the servlet standard. It serves as a starting point for building your own custom servlets. In addition to this base type, there is an abstract base class called `HttpServlet`[4] that plays a crucial role in simplifying the development of servlets. This class already includes prebuilt functionality for handling HTTP methods and directing requests to specific Java methods. This implementation of the template design pattern can be a powerful tool for developing efficient and effective servlets.

There are different ways to register servlets in Spring, two of which are presented now.

@WebServlet with urlPatterns and @ServletComponentScan

An example of a servlet is quickly written:

```
import jakarta.servlet.annotation.WebServlet;
import jakarta.servlet.http.*;
import …

@WebServlet( urlPatterns = "/stat" )
```

---

[3] *https://jakarta.ee/specifications/servlet/5.0/apidocs/jakarta/servlet/servlet*

[4] *https://jakarta.ee/specifications/servlet/5.0/apidocs/jakarta/servlet/http/httpservlet*

```
public class StatisticServlet extends HttpServlet {

  @Autowired …;

  @Override protected void service( HttpServletRequest  req,
                                    HttpServletResponse resp )
      throws IOException {
    resp.getWriter().write( "123456" );
  }
}
```

The class is annotated with `@WebServlet`[5] and defines the path it will eventually take. None of the servlet data types come from Spring; they are pure Jakarta types.

For a call that occurs under the named path, the servlet container calls the `service(…)` method and passes an `HttpServletRequest`[6] and an `HttpServletResponse`[7] object. Within a servlet, the request object provides valuable information about the incoming request from the client. This information can be leveraged to access request cookies or the request body, among other things. On the other hand, the response object enables the program to generate a response that can be sent back to the client. For instance, in the example provided, the response object is used to write a basic string response. By using the response object, developers can control the content that is sent back to the client in response to their request.

This servlet isn't automatically detected and registered as a Spring-managed bean because something is missing for that. If this is desired, at any `@Configuration` the annotation `@ServletComponentScan`[8] must be set on any `@Configuration`. Then, any `@WebServlet` annotated class will be recognized in the search packages, and other Spring-managed beans can be injected so that the servlet can access services or repositories.

The option to set the path at `@WebServlet` has two disadvantages: First, the servlet must know which path it takes, and second, the path is encoded in the annotation, so it can't be dynamic. Therefore, there is a second way.

ServletRegistrationBean[9]

The second option is a servlet registration bean:

---

[5] *https://jakarta.ee/specifications/servlet/5.0/apidocs/jakarta/servlet/annotation/webservlet*

[6] *https://jakarta.ee/specifications/servlet/5.0/apidocs/jakarta/servlet/http/httpservletrequest*

[7] *https://jakarta.ee/specifications/servlet/5.0/apidocs/jakarta/servlet/http/httpservletresponse*

[8] *https://docs.spring.io/spring-boot/docs/current/api/org/springframework/boot/web/servlet/ServletComponentScan.html*

[9] *https://docs.spring.io/spring-boot/docs/current/api/org/springframework/boot/web/servlet/ServletRegistrationBean.html*

```
@Bean
public ServletRegistrationBean statServlet() {
  return new ServletRegistrationBean(
      new StatisticServlet(), "/stat" );
}
```

Then the `@WebServlet` annotation is no longer necessary for the `StatisticServlet`, and the `@ServletComponentScan` annotation can also be omitted.

9.2.4 Weaknesses of Servlets

As we've just learned, servlets form the backbone of most web applications built in Java. However, the servlet application programing interface (API) is considered to be at a low level of abstraction, and modern web applications require additional functionality to meet the needs of developers. Some of the essential features that are required for modern web applications include easy access to path variables in the URL, validation of transferred values, conversion of data to and from the JSON format for transfer objects, content negotiation, the ability to use a template engine for HTML templates, mapping of request data to Java objects, support for multipart file uploads, and easy testing options through mocks.

When compared to this list of features, the basic servlet API falls short. While servlets themselves are a foundational technology that every Java web server implements, programming large-scale web applications exclusively with servlets can be cumbersome and time-consuming. To address this issue, developers need a convenient framework that builds upon the servlet standard and provides these additional features.

9.3 Spring Web MVC

Spring defined not one, but two different web frameworks:

- Spring Web MVC
- Spring WebFlux

Spring WebFlux uses a reactive approach, where the programming model and API are comparable to Spring Web MVC. However, internally, it's structured differently.

While the two frameworks were developed specifically for Spring, web frameworks from the Jakarta Enterprise Edition (Jakarta EE) can also be integrated with Spring:

- Implementation of *Jakarta RESTful Web Services* (formerly JAX-RS), such as *Jersey* or *RESTEasy*
- Implementation of *Jakarta Faces* (formerly *JavaServer Faces* [JSF]) via *JoinFaces* (*http://joinfaces.org/*)

Because Spring Web MVC is the most important in everyday life, we'll fade out the other technologies.

9.3.1 Spring Containers in Web Applications

So far, we've seen that Spring can contain an embedded web server and can also register servlets. Spring Web MVC takes advantage of this by installing a special servlet called the *dispatcher servlet*. The class is named the same: `DispatcherServlet`.[10] The `DispatcherServlet` is also known as a *front controller* because it's "at the front" and handles all HTTP requests. Then, based on criteria such as paths, HTTP methods, parameters, and so on, the dispatcher servlet will forward the request to a *controller*, as shown in Figure 9.4.

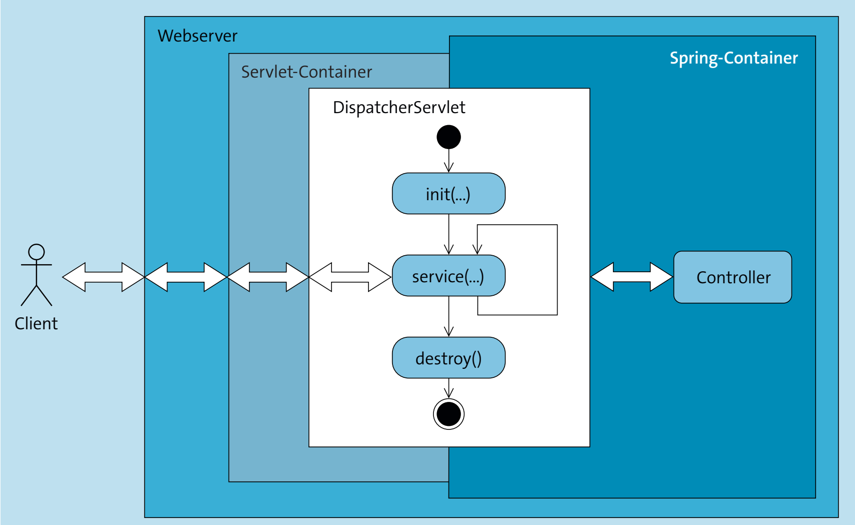


Figure 9.4 "DispatcherServlet" Forwards to Controller

Thus, the dispatcher servlet mediates between the external world and the components in the Spring container. The dispatcher servlet is itself a component in the Spring container. The context in which the beans live is a special context called the *Web Application Context*. The dispatcher servlet is automatically configured via the `DispatcherServletAutoConfiguration` auto-configuration in Spring Boot.

Handler Mapping

When the dispatcher servlet receives an HTTP request, it must decide how to proceed. To accomplish this, the dispatcher servlet uses the *handler mapping*. Handler mappings take over certain resources. For example, a bean named `resourceHandlerMapping` takes over documents in the *static* directory. There is a `RequestMappingHandlerMapping` that takes over classes annotated with `@Controller` or `@RestController`. There are other handlers that take over servlets, for example, or other resources. The Javadoc of `WebMvcConfigurationSupport`[11] explains the order of the handlers.

---

[10] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/servlet/DispatcherServlet.html*

In the text, the annotations `@Controller` and `@RestController` just appeared—we need to take a closer look at them.

9.3.2 @Controller

Annotation `@Controller` is used with classes whenever a method returns a *view ID*. Here's an example:

```
@Controller
public class Date4uWebController {
  @RequestMapping( "/profile" )
  public String profilePage() { return "profile"; }
}
```

The class has a specially annotated method that handles the request and is therefore also called the *handler method*. In this context, the `@RequestMapping` annotation indicates that the controller responds to requests made to the */profile* path. It's worth noting that the method's name isn't significant. However, it's crucial to understand that the method's return value isn't what is sent to the client as a string. Instead, the string is a view ID, which is typically translated into HTML using a template engine. Template engines allow for filling a template with data. Common examples of template engines include *Thymeleaf*, *FreeMarker*, and *Velocity*. Therefore, controllers are used whenever the developer doesn't want to define the response body themselves but wants to forward to a template instead. The template is then filled with form data or other relevant information, and the resulting output is sent back to the client.

@Controller and @ResponseBody

In everyday life, you'll more often find a combination of `@Controller` and `@ResponseBody`. Here's another example:

```
@Controller
// @ResponseBody
public class StatisticRestController {
  @RequestMapping( "/api/stat/total" )
  @ResponseBody
  public String totalNumberOfRegisteredUnicorns() {
```

---

[11] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/servlet/config/annotation/WebMvcConfigurationSupport.html*

```
    return "1234567";
  }
}
```

The use of the `@ResponseBody` annotation in Spring Web MVC signals that the controller manages the response body on its own. As such, the handler method doesn't return a view ID and doesn't invoke a template engine. Instead, it directly returns a string, object, or other data, which can be converted to JSON or XML format as needed.

@ResponseBody

Annotation `@ResponseBody`[12] is allowed locally on methods or also on a type, as shown in the declaration:

```
package org.springframework.web.bind.annotation;

import …

@Target({ElementType.TYPE, ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface ResponseBody { }
```

Defining the `@ResponseBody` annotation locally provides the benefit of allowing some template engine methods to return view IDs while other methods can construct the message body directly. However, if the annotation is applied at the class level, it applies to all methods that define and return the message content. It's worth noting that the `@ResponseBody` annotation doesn't have any attributes.

For `@ResponseBody`, Spring uses `HttpMessageConverter` objects that, based on the content type, convert to formats such as text, JSON, or XML. We'll look at this in more detail in Section 9.7.

9.3.3 @RestController

The use of the `@ResponseBody` annotation isn't very common in code these days. Typically, controllers are designed with handler methods that return the response body directly, meaning that a template engine isn't needed to generate an HTML page. This approach is much more prevalent today than it was 15 years ago, when controllers with template engines were more commonplace. Given how frequently the `@ResponseBody` annotation is used with an `@Controller` annotation on a class, Spring has introduced the composite annotation type `@RestController` to simplify the process:

---

[12] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/bind/annotation/ResponseBody.html*

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Controller
@ResponseBody
public @interface RestController {

        @AliasFor(annotation = Controller.class)
        String value() default "";

}
```

Let's return to the example where `@Controller` and `@ResponseBody` were still separate and use `@RestController` from now on:

```
@RestController
public class StatisticRestController {

  @RequestMapping( "/api/stat/total" )
  public String totalNumberOfRegisteredUnicorns() {
    return "1234567";
  }

}
```

The `StatisticRestController` class serves the purpose of providing statistics on the number of registered unicorns or profiles. Being a `@RestController`, the class is identified through classpath scanning and is registered as a Spring-managed bean.

While the method name `totalNumberOfRegisteredUnicorns` may not be meaningful or relevant, the `@RequestMapping` annotation specifies the URL path that should be associated with the controller and its method. It's worth noting that the visibility of the method isn't a crucial factor, but you should set it as `public`, as recommended by *SonarSource*.[13]

@RestController: An @Controller and an @Component

Each `@RestController` is a Spring-managed bean, and that means we can easily wire it, such as injecting the `ProfileRepository` into the `StatisticRestController`:

```
@RestController
public class StatisticRestController {

  private final ProfileRepository profiles;
```

---

[13] *https://rules.sonarsource.com/java/RSPEC-3751*

```
  public StatisticRestController( ProfileRepository profiles ) {
    this.profiles = profiles;
  }

  @RequestMapping( "/api/stat/total" )
  public String totalNumberOfRegisteredUnicorns() {
    return String.valueOf( profiles.count() );
  }

}
```

From our create, read, update, delete (CRUD) repository, we can fall back on the `count()` method because it gives us the number of entities.

Distinct Responsibilities of a Controller

Controllers in Spring are responsible for accessing the repository or business logic, but they don't implement any business logic themselves. They act as components that understand HTTP and can interpret HTTP headers, path and query parameters, and status codes. They then use this information to make a call to the business logic. However, how the business logic is implemented isn't the controller's concern. For instance, the same business logic could be used for a chat interface. Typically, there is one controller per business transaction or for each business domain. In this example, we'll create different controllers for photos and profiles.

Controllers, like other services in Spring, are *singletons* by default. This means that they exist only once in the application. As multiple HTTP clients can call the same controller endpoint simultaneously, multiple threads may enter the controller instance. Therefore, instance variables within the controller shouldn't be used to "remember" anything for later, as they may be overwritten by another client later.

9.3.4 Controller to [Service] to Repository

The data retrieval process of our `StatisticRestController` directly accesses the repository. However, there are diverging opinions on whether this is a sound practice. One perspective advocates for direct repository access, while another viewpoint suggests that controllers should primarily access a service layer, which in turn interacts with the repository. Figure 9.5 depicts these contrasting approaches.

There are arguments for both approaches, as follows:

- **Controller → Repository**
  When the repository satisfies the controller's requirements precisely, there is no need for a service layer. In such cases, including a service layer would only add redundant repository method instances that are forwarded. This approach helps maintain a streamlined program. However, the challenge arises when the controller's tasks

exceed the repository's capabilities. This is because services typically perform tasks beyond data retrieval. They may aggregate or enrich data from multiple repositories. For instance, when registering a unicorn, the operation entails inserting an entity and sending a confirmation email.

- **Controller → Service → Repository**
  When the controller's requirements exceed the repository's capabilities, a service layer is necessary. This is because business logic shouldn't reside in the controller. If the controller needs both repository and service functionalities simultaneously, the required repository methods should be included in the service. It's not advisable for the controller to directly access both the service and the repository. Instead, it should access them solely through the service layer. Because controllers typically require additional functionality, it's often beneficial to begin with services. Furthermore, to ensure that controllers don't access repositories and that services always act as intermediaries, architecture tests such as those offered by *ArchUnit* (*www.archunit.org*) can be helpful. The *Spring Modulith* project (*https://spring.io/projects/spring-modulith*) is currently being developed for this purpose. It's based on ArchUnit and aims to provide order in monolithic Spring applications.

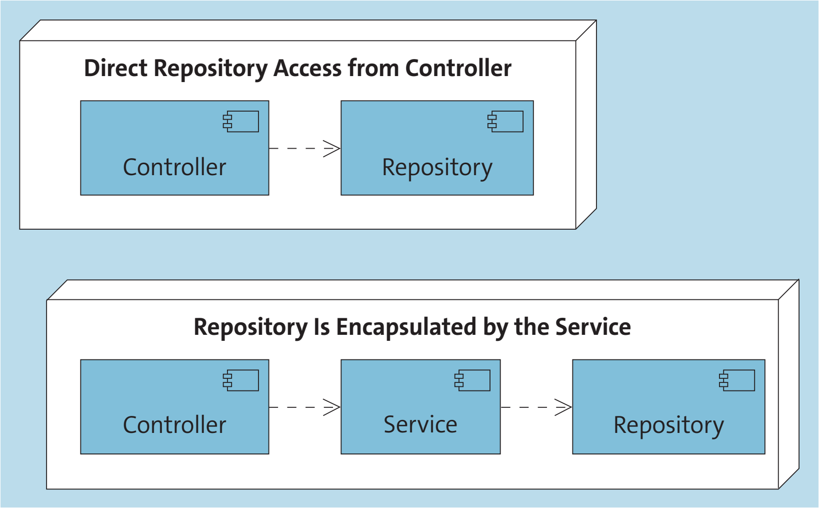

Direct Repository Access from Controller
Controller
Repository
Repository Is Encapsulated by the Service
Controller
Service
Repository

Figure 9.5 Unshielded and Shielded Repository

9.4 Hot Code Swapping

A drawback of Spring application development is that, unlike with an application server, changes made to the source code or bytecode aren't automatically detected after launch. This is because a Spring application typically includes the web server. During development, it's preferable for modifications made to a controller to be immediately noticeable by reloading the page. This process is commonly referred to as *hot code swapping* or *hot swapping*. To accomplish this, there are several approaches. Let's take a look at two.

9.4.1 Hot Swapping of the Java Virtual Machine

The *Java virtual machine* (JVM) supports hot swapping automatically when a program is running in debug mode. In *Eclipse*, this runs more smoothly than in *IntelliJ* because in IntelliJ a build has to be triggered first (`Ctrl` + `F9`). Then you can refresh the browser window (also with `F5`), and a code change of the controller class is visible.

However, the debug mode has a few weaknesses. First, only bytecode within methods[14] is exchanged; no initialization routine runs. This means a change of meta information, such as changed paths to annotations, has no effect. Second, changes to configuration files are also not code and don't cause the bytecode to be replaced.

9.4.2 Spring Developer Tools

An alternative to hot swapping is the *Spring Developer Tools* (dev tools). Here, Spring starts an initial bootstrap program, which in the next step loads the actual application in an extra class loader. This bootstrap class loader registers whether bytecode has been swapped in the background or whether certain central configuration files have been changed. If so, this bootstrap loader will remove the old program and reboot the Spring context. In this case, the entire Spring program is actually restarted, and we lose all volatile state.

If you want to try out the dev tools, all you have to do is include a dependency in the class path:

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-devtools</artifactId>
  <optional>true</optional>
</dependency>
```

Listing 9.2 pom.xml Extension

Another advantage of the development tools is that they can also trigger reloading when selected own resources are changed. The reference documentation[15] explains this in more detail.

Because the dev tools reinitialize the entire context, it takes longer to start than solving via the debugger, but hot swapping through the debugger just can't detect many changes. Teams that want to dig deeper into their pockets can take a look at *JRebel* (*www.jrebel.com/products/jrebel*).

---

[14] Constructors, object initializers, and static initialization blocks are just special methods for the JVM.

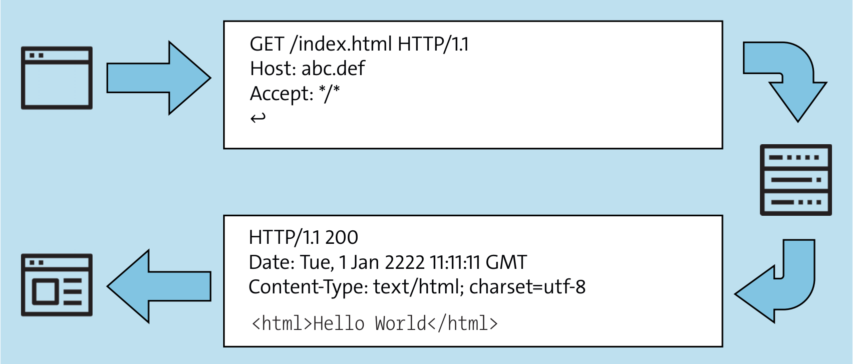
[15] *https://docs.spring.io/spring-boot/docs/current/reference/html/using.html#using.devtools*

9.5 HTTP

Up to this point, we've launched the web server and defined a path that it responds to. As the server needs to send information to and receive data from the client, it's essential to familiarize ourselves with the *HTTP*. Understanding how data is transmitted through HTTP is crucial to gaining a deeper understanding of the entire data exchange process.

9.5.1 HTTP Request and Response

When a client, whether it's a web browser or other software, makes HTTP requests, a small packet of text is built. The server also responds with lines of text, which may be followed by binary data. Figure 9.6 makes this clear.



```
GET /index.html HTTP/1.1
Host: abc.def
Accept: */*
↩
```

```
HTTP/1.1 200
Date: Tue, 1 Jan 2222 11:11:11 GMT
Content-Type: text/html; charset=utf-8
<html>Hello World</html>
```

Figure 9.6 Exchanged Packets in HTTP

- **Client → server**
  The data exchange between the client and the server starts with an HTTP method, which is a *verb* denoting the type of request being made. Apart from the `GET` method, there are other methods such as `POST`, `PUT`, or `DELETE`. This is then followed by a path specification indicating the location of the document on the web server. The HTTP version being used is also included. Next, one or more *HTTP headers* are added, consisting of key-value pairs separated by a colon. Although any headers can be transmitted, there are predefined headers, such as the host being contacted or `Accept`, which specifies the return type the client expects. A line break separates the headers from an empty line, which signifies the end of the headers. The client may then add data to the body, although this is uncommon for `GET` requests and more typical for `POST` and `PUT` requests.
- **Server → client**
  Upon receiving the HTTP packet, the web server proceeds to process the request. The server checks whether it can fulfill the client's request for the document *index.html* using the HTTP method `GET`. If it can, the server sends back a response that follows the format shown in Figure 9.6. The response starts with an identifier indicating the

HTTP version being used, followed by an *HTTP status code* that represents the success or failure of the request. In the given example, the status code is `200`, indicating that everything is working correctly. After the status code, there could still be a string representation such as `OK`, but the example doesn't show that. The server may also send additional headers, which are often standardized. In this case, the server sends a date and a *content type* header, which specifies the format of the data being sent. The content type is also called *MIME type*. The response body contains the requested information, which can be in various formats such as HTML, JSON, or PDF. The web browser will interpret and display the data accordingly based on its content type.

HTTP has been around for a long time and has different versions. Versions 1.1 and 2.0 are widely used, while HTTP/3 is still not very common and is mostly used internally by cloud providers. If you're interested in learning more about the specifics of the newer HTTP, check out *www.rfc-editor.org/rfc/rfc2616*.

9.5.2 Set Up an HTTP Client for Testing Applications

Until now, we've been using the web browser to access our endpoint. While this works well for viewing HTML documents or images, the browser doesn't support HTTP methods such as `PUT` or `DELETE` without the use of a plugin. Additionally, it's not possible to customize the headers that are included in the HTTP message. The headers and status codes that are returned from the server are only visible in the developer console of the browser.

What we need then is an HTTP client that can be used to customize the HTTP messages as desired, and there are various tools on the market, and here's a selection:

- Insomnia (*https://insomnia.rest*)
- Postman (*www.getpostman.com*)
- Advanced REST Client (*https://install.advancedrestclient.com*)
- SoapUI (*https://www.soapui.org*)
- IntelliJ Ultimate (*https://www.jetbrains.com/idea/*)
- RESTED Browser Plugin (*https://addons.mozilla.org/firefox/addon/rested*)

IntelliJ HTTP Client

The IntelliJ IDE also appears in the previous list. With the HTTP client plugin, HTTP requests can be created, edited, and executed directly in the IntelliJ editor. Under **File • New • HTTP Client**, a text file with file extension *http* can be created that looks almost like an original HTTP request.

Note

More details about the HTTP client and syntax can be found on the websites *www.jetbrains.com/help/idea/http-client-in-product-code-editor.html* and *www.jetbrains.com/help/idea/exploring-http-syntax.html*.

A query on the `/api/stat/total` endpoint would look like this with the IntelliJ HTTP client:

```
GET http://localhost:8080/api/stat/total
```

The result is as follows:

```
HTTP/1.1 200
Content-Type: text/plain;charset=UTF-8
Content-Length: 2
Date: Tue, 30 May 2023 11:18:27 GMT
Keep-Alive: timeout=60
Connection: keep-alive

20
```

9.6 Request Matching

When a request is received by the server and reaches the front controller in Spring Web MVC, a decision must be made regarding how to handle the path. This can be thought of as a filter, with additional criteria added beyond just the path and HTTP method, such as headers or parameter values.

9.6.1 @RequestMapping

When we wrote the first web service, we learned about the annotation `@RequestMapping`:[16]

```
@RequestMapping( "/api/stat/total" )
// @RequestMapping( path = "/api/stat/total" )
public String totalNumberOfRegisteredUnicorns() …
```

In the simple form, only the path is in the foreground as a filter criterion, and the HTTP method isn't relevant. For the annotation attribute `value`, there is the alias `path`.

With `@RequestMapping`, the HTTP method isn't significant, but normally endpoints should only be accessed with certain HTTP methods. That's why `@RequestMapping` allows defining another annotation attribute `method`:

```
@RequestMapping( path   = "/api/stat/total",
                 method = { RequestMethod.GET } )
public String totalNumberOfRegisteredUnicorns() …
```

---

[16] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/bind/annotation/RequestMapping.html*

The annotation attribute `method` of type `RequestMethod`[17] is assigned an array of possible HTTP verbs, and the handler method is called only when the client arrives with the HTTP method.

9.6.2 @*Mapping

If `@RequestMapping` has only one common `RequestMethod` set, there is a shortcut via the following annotations: `@GetMapping`,[18] `@PostMapping`, `@PutMapping`, `@DeleteMapping`, and `@PatchMapping`. This would make the following spellings equivalent:

```
// @RequestMapping( path = "/api/stat/total",
//                  method = RequestMethod.GET )
@GetMapping( "/api/stat/total" )
public String totalNumberOfRegisteredUnicorns() …
```

The path specification remains.

9.6.3 More General Path Expressions and Path Matchers

Occasionally, you need more general path expressions, and there are various notations and options for these. One option with `@RequestMapping` is to specify an array of paths with `value`:

```
@RequestMapping( {"", "/page", "page"} )
```

In addition, Spring Web MVC provides *path matcher*, a mechanism that links URLs with certain patterns to handler methods. This can be used to catch groups of paths, similar to regular expressions. Here's a simple example:

```
@RequestMapping( "/**/profiles" )
```

The path matcher verifies whether the requested path matches one of the patterns defined in the application controller. If so, the appropriate handler method is called to process the request. The specification in the example also recognizes path such as */api/demo/acme-inc/profiles* or */profiles*. The documentation for the `PathPattern` explains the details.[19,20]

---

[17] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/bind/annotation/RequestMethod.html*

[18] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/bind/annotation/GetMapping.html*

[19] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/util/pattern/PathPattern.html*

[20] `PathPattern` is the successor of `AntPathMatcher`: *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/util/AntPathMatcher.html*.

9.6.4 @RequestMapping on Type

Because there are the concrete `@*Mapping` annotations, you might think that `@RequestMapping` is no longer necessary. However, `@RequestMapping` has an advantage: the annotation is allowed on the type, and then you can set a base path for all handler methods. The special `@*Mapping` annotations are *only* valid on methods.

Suppose a handler method is to take the path */api/stat/total*:

```
@RequestMapping( "/api" )
public class StatisticRestController {

  @GetMapping( "/stat/total" )
  public String totalNumberOfRegisteredUnicorns() { … }
}
```

Then `@RequestMapping( "/api" )` determines that all handler methods in the class are under */api*. The specification at the handler method with the path */stat/total* leads to the desired result. If `@RequestMapping` doesn't specify a path, / automatically applies to the root directory. In principle, you can also omit the path from the handler methods, so that only the HTTP method decides what should happen at the concrete endpoint.

Suppose we aim to create an endpoint with the path */api/stat/total*. There are essentially three ways to achieve this. First,

```
@RequestMapping( "/api" )
public class StatisticRestController {
  @GetMapping( "/stat/total" )
  public String totalNumberOfRegisteredUnicorns() { … }
}
```

or

```
@RequestMapping( "/api/stat" )
public class StatisticRestController {
  @GetMapping( "/total" )
  public String totalNumberOfRegisteredUnicorns() { … }
}
```

or

```
@RequestMapping( "/api/stat/total" )
public class StatisticRestController {
  @GetMapping
  public String totalNumberOfRegisteredUnicorns() { … }
}
```

9.7 Send Response

In the previous example, we returned a string from the handler method. However, it's important to note that handler methods can return a variety of different types, which is what we'll be exploring in this section.

9.7.1 HttpMessageConverter in the Application

The unit that converts the corresponding return types of the handler methods into a target format is called `HttpMessageConverter`. As an introductory example, let's look at a new class, `PhotoRestController` that can be used to retrieve photos from the outside:

```
@RestController
public class PhotoRestController {

  private final PhotoService photos;

  public PhotoRestController( PhotoService photos ) {
    this.photos = photos;
  }

  @GetMapping( path     = "/api/photos",
               produces = MediaType.IMAGE_JPEG_VALUE )
  public byte[] photo() {
    String imagename = "unicorn001";
    return photos.download( imagename ).orElseThrow();
  }
}
```

Listing 9.3 PhotoRestController.java

If you later call *localhost:8080/api/photos/*, you should see a unicorn, at least if an image named `unicorn001`, that is, a file *unicorn001.jpg*, exists in the *fs* file system.

The reason we see this as an image and not an untyped byte stream is that `produces = MediaType.IMAGE_JPEG_VALUE` sets the content type. If the line was missing, Spring would set the following header:

```
Content-Type: application/octet-stream
```

Then, the same data would arrive, but a browser would no longer be able to display an image.

9.7.2 Format Conversions of Handler Method Returns

Whenever a handler method returns a value, Spring searches for a corresponding `HttpMessageConverter`[21] object that takes the data from the method's return value and adds it to the output stream for the client. The `HttpMessageConverter` instances also work in the opposite direction, as controllers can also receive data that needs to be converted into an object. There are quite a few implementations:

- `StringHttpMessageConverter` ✓
- `ByteArrayHttpMessageConverter` ✓
- `ResourceHttpMessageConverter`
- `MappingJackson2HttpMessageConverter`
- `Jaxb2RootElementHttpMessageConverter`

Because Spring used two of the converters in our examples, there is a ? after the classes. `StringHttpMessageConverter` was used by Spring in the first web service when we returned a `String`:

```
@GetMapping(…)
public String totalNumberOfRegisteredUnicorns() …
```

In addition, `ByteArrayHttpMessageConverter` became active here at the byte array of the photo:

```
@GetMapping( … )
public byte[] photo() …
```

9.7.3 Mapping to JSON Documents

Interchange formats such as JSON or XML documents are more useful than simple strings and byte arrays. In Spring Web MVC, if the handler method return type isn't a string, byte array, or `Resource`, JSON is written by default. For returning XML, an additional library dependency is required for mapping purposes.

LastSeenStatistics Class

The *Date4u* application stores various types of data, such as information about when a user's profile was last active. We want to develop a web service that can deliver this specific data. For the data, we build a class as a container, and Spring will then convert the object to JSON on its own. As a container, we declare a class called `LastSeenStatistics`:

```
public class LastSeenStatistics {
  public static class Data {
    @JsonProperty( "x" ) public YearMonth yearMonth;
```

---

[21] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/http/converter/HttpMessageConverter.html*

```
    @JsonProperty( "y" ) public int count;

    public Data() { }
    public Data( YearMonth yearMonth, int count ) {
      this.yearMonth = yearMonth;
      this.count = count;
    }
  }

  public List<Data> data;

  public LastSeenStatistics() { }
  public LastSeenStatistics( List<Data> data ) { this.data=data; }
}
```

Listing 9.4 LastSeenStatistics.java

The parameterless constructor of the container allows for automatic creation by a JSON library, which isn't required in our example because we're only sending objects and not receiving them. The parameterized constructor enables easy creation of objects with data, such as a `LastSeenStatistics` object that consists of a list of `Data` elements. The nested `Data` class includes a `yearMonth` variable to collect statistics for a specific year and month, as well as a counter to track how many profiles were viewed during that month. Because Jackson can also map records, the following is a compact alternative:

```
record LastSeenStatistics( List<Data> data ) {
 record Data(
   @JsonProperty( "x" ) YearMonth yearMonth,
   @JsonProperty( "y" ) int count
 ) { }
}
```

To override the default property names used when mapping to JSON documents, two proprietary JSON-specific annotations, `@JsonProperty`, are added. In this case, the keys should be renamed as `x` and `y` instead of `yearMonth` and `count`. There are other ways to rename properties, such as using *Jackson mixins*. Then, you could omit the annotations on the container and instead introduce a new interface for the record:

```
@JsonMixin( LastSeenStatistics.Data.class )
interface LastSeenStatisticsDataMinxin {
  @JsonProperty( "x" ) YearMonth yearMonth();
  @JsonProperty( "y" ) int count();
}
```

Annotation `@JsonMixin`[22] comes from Spring Boot and automatically registers the mixin with Jackson's `ObjectMapper`. If the container isn't a record but a class, no mixin interface is declared, but an abstract class with annotated object variables is.

Output Statistics in JSON

We can now use the `LastSeenStatistics` class as a return type in a handler method. To demonstrate this, we'll add a new method called `lastSeenStatistics()` to the `StatisticRestController`.

```
@RestController
public class StatisticRestController {

  …

  @GetMapping( path = "/api/stat/last-seen"
               //, produces = MediaType.APPLICATION_JSON_VALUE
             )
  public LastSeenStatistics lastSeenStatistics() {
    return new LastSeenStatistics(
      Arrays.asList( new LastSeenStatistics.Data( YearMonth.now(),
                                                  100 ) ) );
  }

  …
}
```

Listing 9.5 StatisticRestController.java Extension

If we access *http://localhost:8080/api/stat/last-seen*, the (formatted) result is the following:

```
{
  "data" : [ {
    "x" : "2022-07",
    "y" : 100
  } ]
}
```

The content type is set to `application/json`. The specification of `produces = MediaType.APPLICATION_JSON_VALUE` is unnecessary because we already know the logic: If it's not a `string`, a `byte[]`, or a `Resource` object, the object is converted to a JSON output by default. Internally, this is handled by the Jackson library.

---

[22] *https://docs.spring.io/spring-boot/docs/current/api/org/springframework/boot/jackson/JsonMixin.html*

Random Data *

To see a little more data, we could adjust the code a bit:

```
@GetMapping( "/api/stat/last-seen" )
public LastSeenStatistics lastSeenStatistics() {
  YearMonth start = YearMonth.now().minusYears( 2 );
  YearMonth end   = YearMonth.now();

  ThreadLocalRandom rnd = ThreadLocalRandom.current();
  List<Data> data =
      Stream.iterate( start, o -> o.plusMonths( 1 ) )
            .limit( start.until( end, ChronoUnit.MONTHS ) + 1 )
            .map( yearMonth -> new LastSeenStatistics.Data(
                                yearMonth,rnd.nextInt(1000,10000)))
            .toList();

  return new LastSeenStatistics( data );
}
```

Listing 9.6 StatisticRestController.java Extension

The program generates random `Data` objects for the past two years and puts them into the `LastSeenStatistics`, which is then returned.

Extension: Real Data from the Database

Random data is pretty to look at, but nothing beats real data. For this, we need to add a method in the `ProfileRepository`:

```
@Query( value = """
  SELECT YEAR(p.lastseen) AS y, MONTH(p.lastseen) AS m, COUNT(*) AS count
  FROM Profile p
  GROUP BY YEAR(p.lastseen), MONTH(p.lastseen)
""" )
List<Tuple> findMonthlyProfileCount();
```

Listing 9.7 ProfileRepository.java Extension

The data is in the three columns, and this is what the repository method outputs as a list of `Tuple` objects; `Streamable<Tuple>` is a good alternative to `List`. We could, of course, have used a projection to an `Object` array or a constructor expression, but `Tuple` is nice and compact. In Chapter 6, Section 6.6.9, the data type came up for the first time.

In the controller, then comes the query:

```
@GetMapping( "/api/stat/last-seen" )
public LastSeenStatistics lastSeenStatistics() {
  YearMonth start = YearMonth.now().minusYears( 2 );
  YearMonth end   = YearMonth.now();

// updated
List<Data> data = profiles.findMonthlyProfileCount().stream().map(
 tuple -> {
  return new Data(
    YearMonth.of(
      Integer.parseInt(tuple.get("y").toString()),
      Integer.parseInt(tuple.get("m").toString()) ),
    Integer.parseInt( tuple.get( "count" ).toString() ) );
 } ).toList();
//

  return new LastSeenStatistics( data );
}
```

The list of `Tuple` objects is mapped to a list of `Data` objects, and this initializes `LastSeenStatistics`.

Chart.js

Modern frontend design typically relies on JavaScript applications to build web pages, making it uncommon for servers to deliver HTML pages. Instead, servers typically provide REST endpoints for web applications to access and visualize data. To illustrate this, let’s take a look at the *Chart.js* library (*www.chartjs.org*), which can be used to display various types of charts. Chart.js is a user-friendly JavaScript library. The following webpage includes JavaScript code that accesses our endpoint:

```
<!DOCTYPE html><html lang="en">
<head>
<script src="https://cdn.jsdelivr.net/npm/chart.js@3.9.1/dist/chart.min.js">
</script>
<script src="https://cdn.jsdelivr.net/npm/chartjs-adapter-date-fns@3.0.0/dist/
chartjs-adapter-date-fns.bundle.min.js">
</script>
</head><body>
<canvas id="chart" height="100px"></canvas>
<script>
fetch("/api/stat/last-seen")
  .then(response => response.json())
  .then(json =>
    new Chart("chart", {
      type: "bar",
      options: {scales: {x: {type: 'time', time: {unit: 'month'}}}},
      data: {datasets: [{data: json.data, label: "#Last Seen"}]}
    }));
</script>
</body></html>
```

Listing 9.8 src/main/resources/static/chart.html

The data is displayed live as a chart on the *http://localhost:8080/chart.html* page.

Preconfigured JSON Mapper *

Jackson is the JSON library that Spring Boot automatically preconfigures. For JSON mapping, Jackson's `ObjectMapper` is preconfigured with three settings:

- `MapperFeature.DEFAULT_VIEW_INCLUSION` is disabled.
- `DeserializationFeature.FAIL_ON_UNKNOWN_PROPERTIES` is disabled.
- `SerializationFeature.WRITE_DATES_AS_TIMESTAMPS` is disabled.

The reference documentation explains how to customize Jackson at *https://docs.spring.io/spring-boot/docs/current/reference/html/howto.html#howto.spring-mvc.customize-jackson-objectmapper*.

Jackson isn't necessarily the fastest implementation, but it's definitely the most common. Basically, Spring Boot's auto-configuration also supports *Gson* (Google JSON) and *JSON-B* (JSON binding).[23] Gson shows the best performance of the three in benchmarks, but is in "maintenance mode" according to Google. This means that Gson still receives updates, but caution is still advisable, as it's unclear how much maintenance this library will still receive.

Spring Web MVC Support for XML Mappings *

Spring Web MVC supports writing data to XML format as well, which is another important interchange format for web applications. However, to accomplish this, a marshaller is required to convert Java objects to XML documents and vice versa. There are several libraries available for this, but typically two libraries are used and are automatically configured by Spring Web MVC.

First, we can refer to *Jakarta XML Binding* (JAXB), formerly called *Java Architecture for XML Binding*. This is an official standard, and the reference implementation comes from Oracle itself. If you want to use JAXB, enter the following in *POM.xml*:

---

[23] *https://docs.spring.io/spring-boot/docs/current/reference/html/features.html#features.json*

```
<!-- JAXB -->
<dependency>
  <groupId>org.glassfish.jaxb</groupId>
  <artifactId>jaxb-runtime</artifactId>
</dependency>
```

A second and somewhat simpler library comes from the Jackson family:

```
<!-- Jackson XML extension -->
<dependency>
  <groupId>com.fasterxml.jackson.dataformat</groupId>
  <artifactId>jackson-dataformat-xml</artifactId>
</dependency>
```

JAXB is a bit more demanding on the Java class structure, and the root elements must carry the `@XmlRootElement` annotation. Jackson doesn't need this setting.

Content Negotiation *

Thanks to *content negotiation*, it's possible that an endpoint or a handler method can send both formats. At the controller, both are initially listed at `produces`:

```
@*Mapping( path = …,
           produces = {"application/json","application/xml"} )
```

The client can use an `Accept` header to indicate that it wants to use `application/json` or `application/xml` and then gets JSON or XML. Spring Web MVC evaluates the specification and picks out the appropriate `HttpMessageConverter`.

By default, Spring evaluates the header, but a query parameter is also possible, so that, for example, the following becomes possible: `?format=json`. This can be customized via `ContentNegotiationConfigurer`.

9.7.4 ResponseEntity = Statuscode + Header + Body

Until now, we've focused on sending the body of an HTTP response and using the `produces` attribute to specify the response body's format. However, in addition to the response body, a server may also need to set the status code and headers for the response.

HTTP Status Code

A status code is a crucial piece of information that informs the client about the success or failure of a web service operation. It lets the client know if the operation was executed properly and if there is a valid result in the body, or if there are issues such as missing permissions or the resource has been moved. Websites such as *https://httpstatusdogs.com* and *https://http.cat/* provide a humorous way of understanding the meaning of each status code. For now, we'll focus on 200 (OK) and 404 (NOT FOUND), and explore other HTTP status codes in Section 9.11.

Spring provides two ways to set the status code. The first is via the `@ResponseStatus` annotation,[24] for example:

```
@PostMapping( "/create" )
@ResponseStatus( HttpStatus.CREATED )
public String create() …
```

The annotation attribute is a constant from the `HttpStatus` enumeration type.[25] The usual status codes are provided.

The status code can't always be set statically, and often the status code isn't fixed until runtime. For example, if an operation failed, "Created" wouldn't be correct, but perhaps "Bad Request" or "Not Found".

ResponseEntity

In Java, a handler method is expected to have three returns for the body, status code, and header. However, this isn't possible in Java. To get around this limitation, we can use a trick and create a container to hold all three values. This container is known as `ResponseEntity`.

Here's a short example of what that looks like:

```
public ResponseEntity<?> handlermethod() {
  Object body = …
  HttpHeaders responseHeaders = new HttpHeaders();
  responseHeaders.…
  return ResponseEntity.status( HttpStatus.ABC )
                       .headers( responseHeaders )
                       .body( body );
}
```

There are several ways to build a `ResponseEntity` object. Here we use a static factory method. The `status(…)` method is passed the status code; this is the same enumeration element as `@ResponseStatus`. Because `status(…)` returns a `ResponseEntity.BodyBuilder`, operations can be cascaded. With the method `headers(…)`, headers can be set or not set. Finally, the message is filled with a body. The body can be `null`, that is, empty.

---

[24] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/bind/annotation/ResponseStatus.html*

[25] *https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/http/HttpStatus.html*

The `ResponseEntity` object can be built dynamically depending on, for example, what response comes from the business logic. It might look like this:

```
if ( all_clear )
 return ResponseEntity.ok( body );
if ( there_was_a_problem )
  return ResponseEntity.status( HttpStatus.BAD_REQUEST )
                       .body( null );
```

Due to certain conditions, the status code, headers, and body can be entirely different. (Exceptions will generally not be queried and translated into status codes. There are other mechanisms for this, which we'll discuss in Section 9.10.)

Let's take a look at Figure 9.7, which is a UML diagram showing the methods.

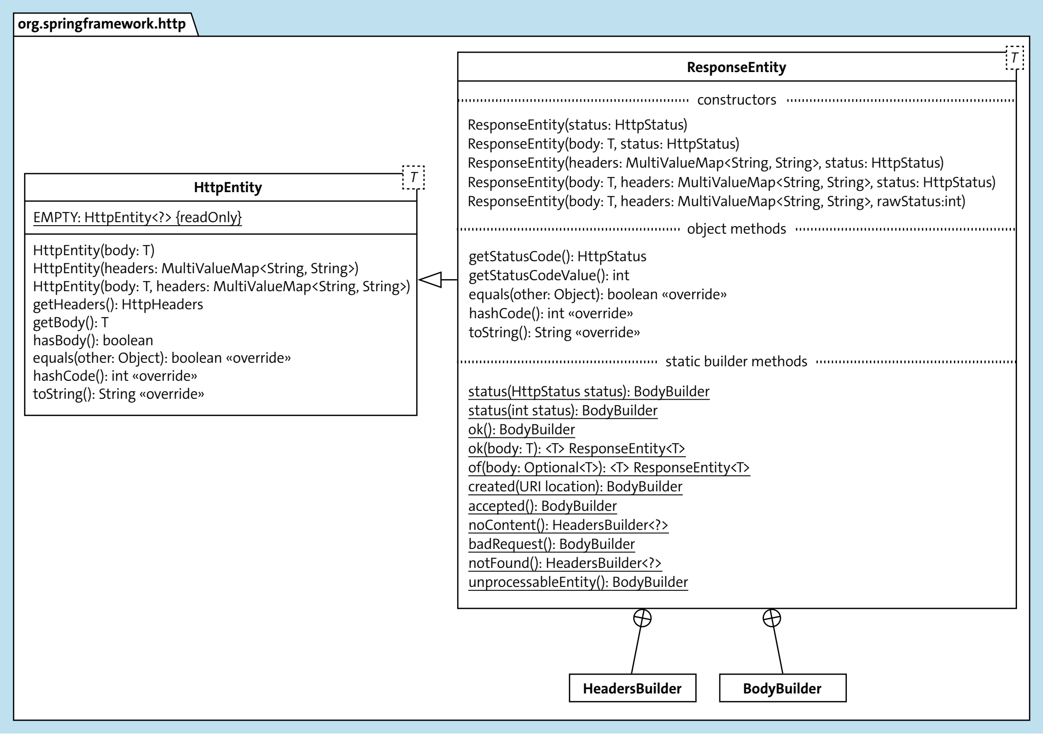

org.springframework.http
ResponseEntity
T
constructors
ResponseEntity(status: HttpStatus)
ResponseEntity(body: T, status: HttpStatus)
ResponseEntity(headers: MultiValueMap<String, String>, status: HttpStatus)
ResponseEntity(body: T, headers: MultiValueMap<String, String>, status: HttpStatus)
ResponseEntity(body: T, headers: MultiValueMap<String, String>, rawStatus:int)
object methods
getStatusCode(): HttpStatus
getStatusCodeValue(): int
equals(other: Object): boolean «override»
hashCode(): int «override»
toString(): String «override»
static builder methods
status(HttpStatus status): BodyBuilder
status(int status): BodyBuilder
ok(): BodyBuilder
ok(body: T): <T> ResponseEntity<T>
of(body: Optional<T>): <T> ResponseEntity<T>
created(URI location): BodyBuilder
accepted(): BodyBuilder
noContent(): HeadersBuilder<?>
badRequest(): BodyBuilder
notFound(): HeadersBuilder<?>
unprocessableEntity(): BodyBuilder
HttpEntity
T
EMPTY: HttpEntity<?> {readOnly}
HttpEntity(body: T)
HttpEntity(headers: MultiValueMap<String, String>)
HttpEntity(body: T, headers: MultiValueMap<String, String>)
getHeaders(): HttpHeaders
getBody(): T
hasBody(): boolean
equals(other: Object): boolean «override»
hashCode(): int «override»
toString(): String «override»
HeadersBuilder
BodyBuilder

Figure 9.7 "ResponseEntity" Class with Its Constructors and Factory Methods

`ResponseEntity` can be created in two ways: using various constructors or static factory methods. The factory methods return a `HeaderBuilder` and require a completion method to create a `ResponseEntity` object.

One interesting factory method is `of(Optional)`. Its logic is as follows: if the `Optional` is empty, the status code is set to "Not Found", and the body is `null`. If the `Optional` isn't empty, the status code is set to “OK”, and the `Optional`’s content is used as the response body.

`ResponseEntity` is a subclass of the `HttpEntity` superclass, which has another subclass called `RequestEntity` that contains information about the request.

9.8 Evaluate Request

After exploring how the server can send data to the client, let’s shift our focus to how the client can transmit data to the server.

9.8.1 Handler Methods with Parameters

In principle, there are two techniques for how a Spring program can get data:

- There are “global objects” (strictly speaking, objects as thread-local variables), for example, `SecurityContextHolder`.
- In general, we’ll indicate the “wish” by a parameter in the parameter list.

A Spring Web MVC handler method can retrieve data from an HTTP message using parameters. There are numerous data types available, and here is a small example:

```
@RestController
public class DemoRestController {
  @RequestMapping( "/api" )
  public void handleRequest( HttpServletRequest  request,
                             HttpServletResponse response ) {
    var requestUri  = request.getRequestURI();
    var requestUrl  = request.getRequestURL();
    var servletPath = request.getServletPath();
    …
  }
}
```

The handler method “wishes” access to the underlying `HttpServletRequest` and `HttpServletResponse` objects in Spring Web MVC. When Spring Web MVC identifies the request, it retrieves the objects and provides them to the handler method. While the two servlet objects are low level, they may be required if the request URL or path is required.

If you’re interested in the complete list of all possible HTTP parameters, visit *https://docs.spring.io/spring-framework/docs/current/reference/html/web.html#mvc-ann-arguments*. There you can read about 30 different data types that can all appear in a parameter list.

9.8.2 Data Transmission from Client to Controller

As a server, it's essential to be able to receive data from the client. To achieve this, it's necessary to understand the various ways in which HTTP allows the client to send data to the server. There are four ways in total, and the following example illustrates them:

```
PATCH /api/profiles/12?validate=none&fast=true   ❶ ❷
Accept: application/json                         ❸
Authorization: Basic ZmlsbG1vcmU6aWFtbm90ZmF0

{ "nickname": "BitterCutiePie" }                 ❹
```

❶ First, data can be part of the path as *path variables*. In the example shown, it's the ID of a profile that is to be changed via the `PATCH` method.

❷ After the path, there can be *query strings* separated by a question mark, which consist of *query parameters*. These key-value pairs are separated by the equal sign, and multiple key-value pairs are separated by an `&` sign.

❸ While HTTP header keys are usually standardized, the value associated with them can be arbitrary. A typical example is a security token.

❹ The client can pass any type of data in the body of the HTTP request, including JSON documents and binary files. This feature provides a significant advantage as the body can handle arbitrarily large data, which isn't possible with path variables, query variables, or header values.

9.8.3 Data Acceptance via Parameters

The Spring Framework decomposes an HTML message into different components, and a handler method can receive these components sent by the client in parameters. Each parameter can be annotated to specify where the value should come from (see Table 9.1).

Desired Access to . . .	Parameter Variable Is Annotated with . . .
Path variable	`@PathVariable`
Query parameters	`@RequestParam`
Header values	`@RequestHeader`
Body	`@RequestBody`

Table 9.1 Annotations Identify the Source of the HTTP Data

According to the table, we can determine how to receive the different components of an HTTP message in a handler method. To get the value of a path variable, we can annotate a parameter variable with `@PathVariable`, for example When the handler method is called by Spring, we can receive the desired path variable in the annotated parameter variable. Let's go through each of the possibilities.

9.8.4 Evaluate Query Parameters

For query parameters, key-value pairs are appended to the path after a question mark. In the following example, on the server, there is an endpoint under the `/profile/search` path (the HTTP method isn't relevant), and there is a query parameter `q=fillmore`:

```
// https://unicorns.example.com/profile/search?q=fillmore
@GetMapping( "/profile/search" )
public String search( @RequestParam String q ) …
```

When a query parameter is passed in the URL, Spring Web MVC assigns it to the parameter variable annotated with `@RequestParam`. For instance, if a request is made to `/profile/search?q=fillmore`, Spring Web MVC breaks down the URL and invokes the `search("fillmore")` method on the controller, with `"fillmore"` being assigned to the `q` parameter, so this will be available in the method.

It's not entirely coincidental that the parameter variable `q` is called the same as the query parameter `q`. But `q` isn't really "speaking," so `query` would be better. But because no assignment is possible anymore and because several query parameters are also possible, an annotation attribute is set:

```
public String search( @RequestParam( "q" ) String query )
```

9.8.5 Optional Query Parameters

Unlike in JavaScript, when dealing with methods in Java, you can't call them with more or fewer arguments. This poses a problem when a query parameter is missing from the URL, and then an argument is missing, leading to an error. To avoid this issue, optional query parameters can be used, allowing for missing parameters. There are three options to choose from.

- **Option 1**: `required = false`, the passed value is `null`:

  ```
  @*Mapping( "help" )
  public String help( @RequestParam( value = "page", required = false )
                      String page ) …
  ```

  We know the notation `required = false` from `@Autowired`, but there are also optional wirings. If `page` isn't specified, the parameter variable remains `null`. However, because `null` quickly leads to an exception, Spring supports the `Optional` data type in many places.

- **Option 2**: Use `Optional`:

  ```
  @*Mapping( "help" )
  public String help( @RequestParam( "page" ) Optional<String> page ) …
  ```

  The `Optional` type isn't as common in the parameter list, but it's okay here. So, `page` will never be `null`, but without a value, it will be `Optional.empty()`.

- **Option 3:** Specify a default value:

  ```
  @*Mapping( "help" )
  public String help( @RequestParam( value="page", defaultValue = "index" )
                      String page ) …
  ```

  The handler method will basically have a value in the parameter variable `page`. It comes either via the passed value, or, if nothing was passed, automatically, as in this example `index`.

9.8.6 Map All Query Parameters

If the number or the names of the query parameters are unknown, all of them can also be put into a data structure:

```
@*Mapping( … )
String foo( @RequestParam MultiValueMap<String,String> params ) …
```

The same key can occur multiple times, and a specification of `q=fillmore&q=bold` is valid. Therefore, we can use the special data structure `MultiValueMap`, although `Map` is also possible.

Incidentally, HTML form data can also be collected in this way. If there is a form, and the page is sent, the key-value pairs are automatically collected by the browser and, for example, linked to a URL or sent in the body. With `MultiValueMap`, all data can be collected and processed automatically.

Task: Last Seen in a Certain Area

We had programmed a handler method for the *api/stat/last-seen* endpoint and set `start` and `end` to a predefined value there:

```
@GetMapping( "/api/stat/last-seen" )
public LastSeenStatistics lastSeenStatistics() {
  YearMonth start = YearMonth.now().minusYears( 2 );
  YearMonth end   = YearMonth.now();
  …
}
```

The task is to adapt the program so that the start and end values are both optional, that is, they can be present or missing. If they are missing, then the current default values are to apply. Thus, four cases must be considered.

Proposed solution: As the parameters aren't mandatory, we must use either use `required=false` or `Optional` in the parameter list. Using `Optional` is beneficial because it enables functional programming.

```
public LastSeenStatistics lastSeenStatistics(
    @RequestParam( "start" ) Optional<String> maybeStart,
    @RequestParam( "end" )   Optional<String> maybeEnd ) {
  YearMonth start = maybeStart.map( YearMonth::parse )
                              .orElse( YearMonth.now().minusYears( 2 ) );
  YearMonth end   = maybeEnd.map( YearMonth::parse )
                            .orElse( YearMonth.now() );

  …
}
```

Listing 9.9 StatisticRestController.java Extension

The identifier names `maybeStart` and `maybeEnd` express that no values must be present, but because the query parameters shouldn't be named `maybe*`, there is an explicit naming with `@RequestParam`.

At the end, there should be `YearMonth` objects. The logic is as follows: If the `Optional` contains a `String`, it's mapped to an `Optional<YearMonth>` with `YearMonth.parse(String)` (or there is a parser error, then the client is out of luck). If the `Optional` was empty, then the default values are taken.

By the way, if you want to work with the real data, you can add a method in the repository:

```
@Query(value = """
  SELECT YEAR(p.lastseen) AS y, MONTH(p.lastseen) AS m, COUNT(*) AS count
  FROM   Profile p
  WHERE  p.lastseen > :startDate AND p.lastseen < :endDate
  GROUP BY YEAR(p.lastseen), MONTH(p.lastseen)""")
List<Tuple> findMonthlyProfileCount( LocalDateTime startDate,
                                     LocalDateTime endDate );
```

The method can be called as follows:

```
profiles.findMonthlyProfileCount(
  start.atDay(1).atStartOfDay(),
  end.atEndOfMonth().plusDays(1).atStartOfDay()
).…
```

Tip

The parameter variable can be of type `String` or `Optional<String>`, but it's also possible to directly use `YearMonth` or `Optional<YearMonth>` types. The conversion of these types is performed by the `ConversionService`, which will be discussed in detail in Section 9.9.

9.8.7 Evaluate a Path Variable

We've discussed only one of the four ways in which a client can pass data to the server, that is, query parameters. Now, let's focus on the second possibility, which is passing data through *path variables*. Path variables are commonly used in resource identification because they are a part of the URL and not an additional "attachment" such as query parameters. To capture and hold the dynamic portion of the URL in a mapping annotation, Spring Web MVC provides a special notation. Here's an example:

```
@GetMapping( "/api/profiles/{id}" )
public Profile get( @PathVariable long id ) …
```

Curly brackets define the path variable, in our case, `id`. The notation with the curly braces is standardized and is called a *Uniform Resource Identifier (URI) template*.[26]

For the path variables, the call works like it does for query parameters: Spring breaks the HTTP message into the appropriate components, calls the handler method, and passes the requested values. Therefore, the path variable `id` is followed by the annotation `@PathVariable`.[27]

If the name of the path variable and the parameter variable differ, an annotation attribute must be set:

```
@GetMapping( "/api/profiles/{id}" )
public Profile get( @PathVariable("id") long profileId ) …
```

Task: Load Images

Recall the `PhotoRestController`, which returned a byte array containing a predefined image. Here is an example of the class:

```
@RestController
public class PhotoRestController {

  private final PhotoService photos;

  public PhotoRestController( PhotoService photos ) {
    this.photos = photos;
  }

  @GetMapping( path     = "/api/photos",
               produces = MediaType.IMAGE_JPEG_VALUE )
  public byte[] photo() {
    String imagename = "unicorn001";
```

---

[26] Described in RFC 6570 at *www.rfc-editor.org/rfc/rfc6570*.

[27] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/bind/annotation/PathVariable.html*

```
    return photos.download( imagename ).orElseThrow();
  }
}
```

Task: Rewrite the method so that you can load images:

- *http://localhost:8080/api/photos/unicorn001* → The image is displayed.
- *http://localhost:8080/api/photos/Nimitz_UFO_Encounters* → The body is `null`, and the status code is 404.

The path *api/photos* should be used to evaluate the image name. If the image exists, it should be returned. However, if there is no photo with that name, the body should be empty, and a **Not Found** (404) status code should be returned.

Proposed solution: The `PhotoService` has already been injected into the `PhotoRestController`, so loading is prepared. We just need to include the path variable in the URI template and add a parameter variable:

```
@GetMapping( path     = "/api/photos/{name}",
             produces = MediaType.IMAGE_JPEG_VALUE )
public ResponseEntity<?> photo( @PathVariable String imagename ) {
  return ResponseEntity.of( photoService.download( imagename ) );
}
```

Listing 9.10 PhotoRestController.java Extension

Method `ResponseEntity.of(Optional)` is useful because it performs the requested translation directly. If the `PhotoService` returns an `Optional.empty()` when trying to load, the body is empty, and the status code is 404. If a loaded byte array comes back, it goes into the body, and the status code is 200.

Multiple Variables Annotated with @PathVariable

While the client can pass only one body, it can pass multiple query parameters, multiple path variables, and multiple headers as well. A URI template can contain multiple path variables, as the following example shows:

```
@GetMapping( "/api/profiles/{id}/photos/{index}" )
public ResponseEntity<?> get( @PathVariable long id,
                              @PathVariable int  index )
```

The objective of this method is to retrieve the photo from the collection associated with the endpoint's profile ID at index 0, 1, 2, and so on. Unlike before, the image name isn't used, so two pieces of information are required. Although it's possible to have multiple path variables and query parameters simultaneously, it's uncommon to have more than two path variables.

Of course, multiple path variables and multiple query parameters are allowed at the same time, but more than two path variables are uncommon.

Task: Create a new class called `ProfileRestController`, and implement the mentioned method.

Proposed solution: First, we need to create the `ProfileRestController` class, and we can then set the path at the class:

```
@RestController
@RequestMapping( "/api/profiles" )
public class ProfileRestController {
}
```

Listing 9.11 ProfileRestController.java

To load a profile with an ID, we need the `ProfileRepository`, and to have the photo loaded later, we need `PhotoService`. Let's inject both:

```
final ProfileRepository profiles;
final PhotoService photos;

public ProfileRestController( ProfileRepository profiles,
                              PhotoService photos ) {
  this.profiles = profiles;
  this.photos = photos;
}
```

Listing 9.12 ProfileRestController.java Extension

Let's get to the heart of the method:

```
@GetMapping( path = "{id}/photos/{index}",
             produces = MediaType.IMAGE_JPEG_VALUE )
public ResponseEntity<?> get( @PathVariable long id,
                              @PathVariable int index ) {

  Optional<Profile> maybeProfile = profiles.findById( id );
  if ( maybeProfile.isEmpty() )
      return new ResponseEntity<>( HttpStatus.NOT_FOUND );

  Profile profile = maybeProfile.get();
  if ( index >= 0 && index < profile.getPhotos().size() ) {
      Photo photo = profile.getPhotos().get( index );
      Optional<byte[]> download = photos.download( photo );
      return ResponseEntity.ok( download.get() );
  }
  return new ResponseEntity<>( HttpStatus.NOT_FOUND );
}
```

Listing 9.13 ProfileRestController.java Extension

Initially, the method attempts to load the profile. If there is no profile found for the given ID, the process terminates and returns a status of **Not Found**. After finding the profile, the method checks whether the index for the associated images is valid. If the index is valid, the method obtains the photo name and forwards it to the `PhotoService`. However, if the index isn't valid, the response has a status code of 404. It's worth noting that multiple path variables and query parameters are allowed, although more than two path variables aren't commonly used.

9.8.8 MultipartFile

For uploading one or more files via a form, there is a separate possibility in HTML. For this purpose, an HTML `form` tag is prepared with a `POST` method. A form element `input type="file"` with the attribute `enctype="multipart/form-data"` is inserted there. Optionally, `multiple` can follow if multiple files can be uploaded.

Spring Web MVC declares the type `MultipartFile`[28] to accept such uploaded files.

Let's assume that a new photo is to be uploaded for a certain profile:

```
@RequestMapping( "/api/profiles" )
public class ProfileRestController {
  …
  @PostMapping( "{id}/photos/" )
  public ResponseEntity<?> saveImage(
    @PathVariable long id,
    @RequestParam MultipartFile file ) { … }
}
```

On a web page, by default, you have a button that opens a file dialog; there's also a fancier drag-and-drop variant.

We can look at how the data is transferred in HTTP and use IntelliJ's HTTP client for uploading:

```
POST http://localhost:8080/api/profiles/3/photos/
Content-Type: multipart/form-data; boundary=boundary

--boundary
Content-Disposition: form-data; name="file"; filename="test.jpg"
```

---

[28] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/multipart/MultipartFile.html*

```
< ./new-unicorn.jpg
--boundary
```

The client uses a special notation with < to allow a file to be uploaded from the file system. The `MultipartFile` data type inherits from `InputStreamSource` (see Figure 9.8), which is a data type we learned about as the base type of `Resource` in Chapter 3, Section 3.5.

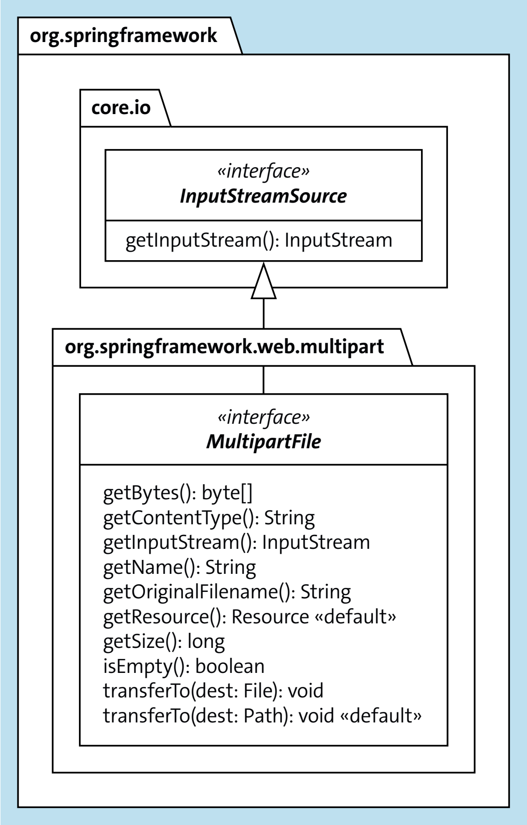

org.springframework
core.io
«interface»
InputStreamSource
getInputStream(): InputStream
org.springframework.web.multipart
«interface»
MultipartFile
getBytes(): byte[]
getContentType(): String
getInputStream(): InputStream
getName(): String
getOriginalFilename(): String
getResource(): Resource «default»
getSize(): long
isEmpty(): boolean
transferTo(dest: File): void
transferTo(dest: Path): void «default»

Figure 9.8 Methods of “MultipartFile”

With `getInputStream()`, a file stream can be opened, and the data can be read. You can see other methods that can redirect the data to a file, for example.

Note

When a `MultipartFile` object is received by the handler method, it’s considered valid only as long as the connection exists. Therefore, it’s crucial to save the data directly as the object is only temporarily valid. If the connection has been closed, it will be too late to perform any actions on the object.

9.8.9 Evaluate Header

Of the four different ways the client can pass data, we've already discussed query parameters and path variables. Now let's look at the headers. The headers sent by the client can be received in a parameter annotated with `@RequestHeader`.[29] Here's an example:

```
public String handlerMethod(
  @RequestHeader("Accept-Language") String acceptLanguage,
  @RequestHeader(value="User-Agent", defaultValue="Bro") String userAgent,
  @RequestHeader("Keep-Alive") long keepAlive
) …
```

We can read that, as with query parameters, default values are also possible and type conversion from string to `long`, for example, works.

All headers can be read into a container of type `Map`, `MultiValueMap`, or `HttpHeaders`:[30]

```
public String allHeaders( @RequestHeader HttpHeaders headers ) …
```

Note

The `Accept-Language` header can be received in the handler methods as a locale object, but this only works if exactly one language is set. Often, however, clients send multiple languages, like this:

```
Accept-Language: de,en-US;q=0.7,en;q=0.3
```

This is where the `HttpHeaders` data type comes in handy because it has the useful methods `List<Locale.LanguageRange> getAcceptLanguage()` and `List<Locale> getAcceptLanguageAsLocales()`, which returns all languages.

9.8.10 HttpEntity Subclasses RequestEntity and ResponseEntity *

We've used the `ResponseEntity` class many times. It has a superclass `HttpEntity`, and this has another subclass, `RequestEntity`[31] (see Figure 9.9).

An HTTP request (`RequestEntity`) and an HTTP response (`ResponseEntity`) have things in common, namely header and body, and the base type `HttpEntity` summarizes them. An `HttpEntity` object is also allowed in the parameter list and as a return.

---

[29] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/bind/annotation/RequestHeader.html*

[30] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/http/HttpHeaders.html*

[31] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/http/RequestEntity.html*

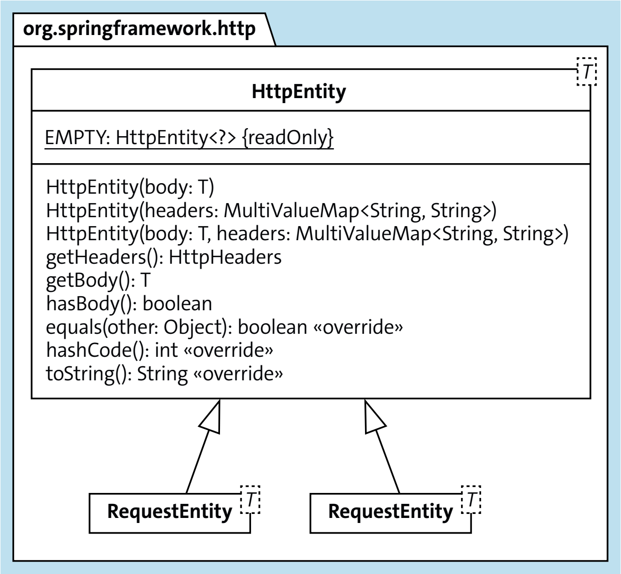

org.springframework.http
HttpEntity
T
EMPTY: HttpEntity<?> {readOnly}
HttpEntity(body: T)
HttpEntity(headers: MultiValueMap<String, String>)
HttpEntity(body: T, headers: MultiValueMap<String, String>)
getHeaders(): HttpHeaders
getBody(): T
hasBody(): boolean
equals(other: Object): boolean «override»
hashCode(): int «override»
toString(): String «override»
RequestEntity
T
RequestEntity
T

Figure 9.9 "HttpEntity" Class with Its Subclasses

9.9 Type Conversion of the Parameters

In this section, we'll look at how Spring Web MVC can convert specific types of parameter lists from handler methods. In Spring, the `ConversionService` helps convert from a source type to a target type. As an example, let's look at the following again:

```
@GetMapping( "/api/profiles/{id}" )
public Profile get( @PathVariable long id ) …
```

The type of the parameter variable isn't `String`, although a URL is of course a string. But the `ConversionService` does the conversion from `String` to `long`.

Because the `ConversionService` can map numerous types, it's likely that the usual data types will work: `long`, `int`, `LocalDate`, and so on. If it gets too exotic, there is an exception.

In multiple areas of the Spring Framework, type conversion is performed through the `ConversionService`. Spring Web MVC leverages this conversion service to handle all annotated parameter variables, including those marked with `@RequestParam`, `@RequestHeader`, `@PathVariable`, `@MatrixVariable`, and `@CookieValue`.

9.9.1 YearMonth Converter Example

Under Section 9.7.3, in the "Output Statistics in JSON" subsection, we developed a web service that accepts a start and end value for the statistic period via query parameters and returns a `LastSeenStatistics` object. For demonstration purposes and for easy testing, we want to simplify the method a little to illustrate the type conversions:

```
@GetMapping( … )
public String lastSeenStatistics(
    @RequestParam( "start" ) Optional<YearMonth> maybeStart,
    @RequestParam( "end" )   Optional<YearMonth> maybeEnd ) {
//  YearMonth start = maybeStart.orElse( YearMonth.now().minusYears( 2 ) );
//  YearMonth end   = maybeEnd.orElse( YearMonth.now() );
  return maybeStart + " " + maybeEnd;
}
```

The return type is now a string and not our container. Moreover, the parameter type is no longer `Optional<String>`, but an `Optional<YearMonth>` object because `YearMonth` objects are considered by the `ConversionService` just like an `int`, `long`, or `double`.

The following HTTP calls return the specified results:

- *http://localhost:8080/api/stat/last-seen* → `Optional.empty Optional.empty`
- *http://localhost:8080/api/stat/last-seen?start=2024-01* → `Optional[2024-01] Optional.empty`
- *http://localhost:8080/api/stat/last-seen?start=2024-01&end=2033-01* → `Optional[2024-01] Optional[2033-01]`
- *http://localhost:8080/api/stat/last-seen?start=2024-01&end=XXXX-XX* → Whitelabel error page

Spring Web MVC automatically calls the `parse(…)` method of `YearMonth` and returns a `YearMonth` object at the end. If `parse(…)` throws an exception, the web service also terminates.

9.9.2 @DateTimeFormat and @NumberFormat

When converting, the formats must be known and it must be clear to the callers that the combination of year and month must have the format “four-digit year + month.” However, there are transfers where different notations are possible. Typical examples are date values and floating-point values.

Date Converter and @DateTimeFormat

For date values, Java offers different data types: `Date`, `LocalDate`, and `LocalDateTime`, to name the most important ones. The `ConversionService` can convert from a `String` to a `Date`, `LocalDate`, or `LocalDateTime`, but the format must conform to ISO-8601 notation. However, you may want to express a date in a shorter way or swap the year, month, or day. With the annotation `@DateTimeFormat`,[32] you can control the format individually so that a mapping still succeeds. `@DateTimeFormat` has four annotation attributes:

- `String pattern`
- `DateTimeFormat.ISO iso`
- `String style`
- `String[] fallbackPatterns`

This can be used to control the structure of the string more precisely so that it's parsed with this pattern. Let's say we have a `LocalDate` object in the parameter list, and the specification should be like this: Month, Day, Year. Then annotation `@DateTimeFormat` could determine the pattern:

```
( @DateTimeFormat(pattern = "MM-dd-yyy") LocalDate date )
```

@NumberFormat

In `org.springframework.format.annotation`, there is another annotation type besides `@DateTimeFormat`, namely `@NumberFormat`,[33] which is used comparable to `@DateTimeFormat`. `@NumberFormat` can be used with all numbers—and a number in Spring is everything that is a subclass of `Number`, such as `Integer`, `Double`, and `BigDecimal`. Annotation type `@NumberFormat` has two annotation attributes:

- **`pattern`**
  This is the formatting string of the number.
- **`style`**
  `DEFAULT`, `NUMBER`, `PERCENT`, and `CURRENCY` are available for selection.

9.9.3 Map Query Parameters and Form Data to a Bean

We've defined two query parameters for determining the minimum and maximum values. Although using two parameters to handle two query parameters is reasonable, it can become overwhelming when dealing with numerous query parameters in a search. For instance, when creating a product page, there may be restrictions by manufacturer, price range, availability, delivery time, and more. If all of these parameters are included in the method, it can be challenging to keep track of them all.

Spring Web MVC offers a better option: it can collect the data and map it to a JavaBean. This internal conversion is done by a `WebDataBinder`; we learned about the data type earlier in Chapter 3, Section 3.6.7. In this context, validations are also performed, and we can react to errors.

---

[32] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/format/annotation/DateTimeFormat.html*

[33] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/format/annotation/NumberFormat.html*

JavaBean YearMonthRange for Mapping

Let's stay with the example of the start and end value for the statistics. The values are to be transferred to JavaBean `YearMonthRange`:

```
class YearMonthRange {
  private String start, end;

  public String getStart() {
    return start;
  }

  public void setStart( String start ) {
    this.start = start;
  }

  public String getEnd() {
    return end;
  }

  public void setEnd( String end ) {
    this.end = end;
  }

  @Override public String toString() {
    return "YearMonthRange[start='%s', end='%s']"
           .formatted( start, end );
  }
}
```

@ModelAttribute

The parameter list no longer has to take the start and end values individually, but can "wish" for the data in the container. For this purpose. only `@ModelAttribute`[34] must be set in front of the container:

```
@GetMapping( "/api/stat/last-seen" )
public String lastSeenStatistics(
    @ModelAttribute YearMonthRange range
) {
  return range.toString();
}
```

---

[34] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/bind/annotation/ModelAttribute.html*

With `@ModelAttribute`, Spring Web MVC knows that the query parameter names should be mapped to the corresponding properties of the JavaBean.

The call options are as before:

- *http://localhost:8080/api/stat/last-seen* → `YearMonthRange[start='null', end='null']`
- *http://localhost:8080/api/stat/last-seen?start=123* → `YearMonthRange[start='123', end='null']`
- *http://localhost:8080/api/stat/last-seen?start=123&end=345* → `YearMonthRange [start='123', end='345']`

If the start value or the end value isn't passed, the setter isn't called, and the instance variables remain at the default.

Tip

`@DateTimeFormat` and `@NumberFormat` can also be used inside the JavaBean.

One advantage of using the `@ModelAttribute` annotation is that it automatically performs type conversion. Therefore, when dealing with a `YearMonthRange` object, we don't need to input the start and end values as strings; we can directly use the `YearMonth` data type:

```
class YearMonthRange {
  private YearMonth start, end;
  // Setter + Getter
  @Override public String toString() {
    return "YearMonthRange[start='%s', end='%s']"
             .formatted( start, end );
  }
}
```

The call options are as follows:

- *http://localhost:8080/api/stat/last-seen?start=2024-01* → `YearMonthRange [start='2024-01', end='null']`
- *http://localhost:8080/api/stat/last-seen?start=2024-01&end=XXXX-XX* → Whitelabel error page

If the conversion isn't possible, a web browser displays a *Whitelabel error page*.

BindingResult

Spring Web MVC uses a `WebDataBinder`[35] for converting the data into a JavaBean, which can report errors via a `BindingResult` object.[36] If there is also a `BindingResult` container next to the `@ModelAttribute` in the parameter list, incorrect values won't lead to an exception, but only to an entry in the `BindingResult` container:

```
@GetMapping( "/api/stat/last-seen" )
public String lastSeenStatistics(
     @ModelAttribute YearMonthRange range,
     BindingResult bindingResult
) {
   return range.toString() + "\n" + bindingResult;
}
```

Let's set the start date in the URL correctly, but the end date incorrectly: *http://localhost:8080/api/stat/last-seenMYM?start=2024-01&end=XXXX-XX*. This then no longer causes the handler method to abort; there is no exception. However, `BindingResult` is filled, and the `toString representation` looks like this:

```
YearMonthRange[start='2024-01', end='null']
org.springframework.validation.BeanPropertyBindingResult: ↩
1 errors Field error in object 'yearMonthRange' on field 'end': ↩
rejected value [XXXX-XX]; codes [typeMismatch.yearMonthRange.end, ↩
typeMismatch.end,typeMismatch.java.time. YearMonth, typeMismatch]; ↩
arguments [org.springframework.context.support. ↩
DefaultMessageSourceResolvable: codes [yearMonthRange.end,end]; ↩
arguments []; default message [end]]; default message ↩
[Failed to convert property value of type 'java.lang.String' ↩
to required type 'java.time.YearMonth' for property ↩
'end'; nested exception is org.springframework.core.convert. ↩
ConversionFailedException: Failed to convert from type [java.lang.String] ↩
to type [java.time.YearMonth] for value 'XXXX-XX'; nested exception ↩
is java.lang. IllegalArgumentException: Parse attempt failed for ↩
value [XXXX-XX]]
```

Our code can decide whether to ignore or correct erroneous values or to stop processing.

---

[35] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/bind/WebDataBinder.html*

[36] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/validation/BindingResult.html*

9.9.4 Register Your Own Type Converters

In the previous examples, the `ConversionService` was able to convert strings to `YearMonth` objects. However, it can also perform conversions between custom data types. Let's explore this functionality by refactoring the `YearMonthRange` class into a more concise record:

```
record YearMonthRange( YearMonth start, YearMonth end ) { }
```

The two record components `start` and `end` contain `YearMonth` objects as before. The goal should be that we can request our own `YearMonthRange` object at this point using the `@RequestParam`:

```
@GetMapping( "/api/stat/last-seen" )
String lastSeenStatistics( @RequestParam YearMonthRange range ) {
  return range.toString();
}
```

No `@ModelAttribute` should be used, but the `ConversionService` should take over the conversion of our data type. For this, the `ConversionService` has to get to know our type `YearMonthRange`. There are two possibilities as described next.

Formatter for Spring Web MVC; YearMonthRangeFormatter

For Spring Web MVC to handle our `YearMonthRangeFormatter` as a parameter type, the first thing we need to do is program a `Formatter`, as discussed previously in Chapter 3, Section 3.6.5. A `Formatter` consists of two parts: a `Parser` and a `Printer`. They do the conversion from a string to the desired data type and back.

The source code of the `YearMonthRangeFormatter` looks like this:

```
class YearMonthRangeFormatter implements Formatter<YearMonthRange> {

  @Override
  public String print( YearMonthRange range, Locale __ ) {
    return range.start + "~" + range.end;
  }

  @Override
  public YearMonthRange parse( String s, Locale __ )
      throws ParseException {
    try {
      YearMonth[] startEnd = Pattern.compile( "~" )
                                    .splitAsStream( s )
                                    .map( YearMonth::parse )
                                    .toArray( YearMonth[]::new );
      return new YearMonthRange( startEnd[ 0 ], startEnd[ 1 ] );
    }
    catch ( DateTimeParseException e ) {
      throw new ParseException(e.getMessage(), e.getErrorIndex());
    }
    catch ( ArrayIndexOutOfBoundsException e ) {
      throw new ParseException("Insufficient number of values", 0);
    }
  }
}
```

The `YearMonth` objects should be separated with a serpentine line (tilde, ~) so that you can write, for example, `2020-02~2024-04` if you want to express the range from `2020-02` to `2024-04`.

To use this `Formatter` implementation, there are two approaches: configure the `Formatter` locally at the controller or configure it globally for all controllers and also at places that have nothing to do with controllers at all.

Type Conversion: @InitBinder resp. WebDataBinder (Local)

If we set the `Formatter` locally, we write a new method and annotate it with `@InitBinder`.[37] The method takes a `WebDataBinder` object[38] and can add the new formatter with `addCustomFormatter(…)`. In the code, it looks like this:

```
@RestController
public class MyWebDataBinderRestController {

  @InitBinder
  public void initBinder( WebDataBinder binder ) {
    binder.addCustomFormatter( new YearMonthRangeFormatter() );
  }

  @GetMapping…
}
```

Because the `YearMonthRangeFormatter` is local to this controller, we would have to do the same for other controllers if they were to know the `YearMonthRangeFormatter` as well. That's why there is a second option.

---

[37] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/bind/annotation/InitBinder.html*

[38] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/bind/WebDataBinder.html*

Type Conversion: FormattingConversionService (Global)

For a global `Formatter`, it's sufficient to place it in the context as a Spring-managed bean:

```
@Configuration
class YearMonthRangeFormatterConfig {

  @Bean
  public Formatter<YearMonthRange> yearMonthRangeFormatter() {
    return new YearMonthRangeFormatter();
  }
}
```

When the Spring Boot application starts up, it takes all beans that implement the `Formatter` interface and automatically registers them.

9.9.5 URI Template Pattern with Regular Expressions

Using a `Formatter` to map a string into any type is a convenient feature. However, in certain cases where a path variable needs to be split into several strings, an alternative approach is available that includes validating the structure's correctness.

We've already been using URI templates regularly, which include expressions in curly brackets. In Spring, these are referred to as *URI variables*. The notation can be extended by adding a colon after the URI variable and appending a regex expression:

```
{URI variable:Regex}
```

Suppose a path variable contains a file name consisting of a file name, a period, and a file extension. It might look like this:

```
@RequestMapping( "/{name:[-\\w]+}.{suffix:\\w+}" )
public void handle( @PathVariable String name,
                    @PathVariable String suffix ) …
```

The string contains two URI variables, and the regex expressions act as a filter. When the front controller receives a request, certain filter criteria must apply for a handler method to deliver. Let's compare this with the simple variant:

```
@RequestMapping( "/{name}.{suffix}" )
```

There are no restrictions at all here. The name and the file extension can be arbitrary. But with the regex, a second handler method would be allowed that recognizes, say, a file name or suffix that just *doesn't* consist of word characters `[a-zA-Z_0-9]`. The parameter list of the second handler method would be identical; only the regex expressions would be different—and Spring Web MVC could still match that.

9.10 Exception Handling and Error Message

In previous examples, we learned that we can identify issues within a controller and automatically translate them into various status codes. However, the question remains whether it's appropriate to catch exceptions in the controller and convert them into different status codes. Spring Web MVC provides a solution to handle this scenario, which we'll explore in this section.

9.10.1 Map Exceptions to Status Codes Yourself

Within the controller, exceptions can be handled, of course, and the `ResponseEntity` object can be filled differently:

```
try {
  …
  return ResponseEntity.ok( … );
}
catch ( … ) {
  return ResponseEntity.status( HttpStatus … )
                       .body( … );
}
```

The controller can catch exceptions thrown by the code and translate them into a status code, which is a viable solution in principle. However, this approach also has its drawbacks. First, adding `try-catch` blocks in the controller doesn't necessarily make the code more readable. Additionally, if different controllers query the same business logic that may throw exceptions, implementing `try-catch` handling in each controller would result in code duplication.

Furthermore, the controller needs to have specific knowledge of which business logic errors should be translated into which status codes and error messages. However, it may not be ideal for the controller to have complete knowledge of this information.

9.10.2 Escalates

If we omit the `try-catch` block and "just let it run," any exception thrown within the controller method would propagate to the Spring Framework. This would trigger Spring's built-in exception handling mechanism, which includes various exception resolver classes.

A new controller is to be introduced for different variants of automatic error handling:

```
@RestController
@RequestMapping( "/api/quotes" )
public class QuoteRestController {
  private final static String[] QUOTES = {
      "Date to be known, not to be liked",
      "Dating is all about the chase. It’s fun!",
      "You can’t blame gravity for falling in love"
  };

  @GetMapping
  public String retrieveQuote( int index ) {
    return QUOTES[ index ];
  }
}
```

The handler method should return a quote for an index. The problem is obvious: if no query parameter is passed, an error will occur. Additionally, an exception is thrown when the index is negative or goes beyond the limit.

If you go back to the *http://localhost:8080/api/quotes* endpoint, there is a Whitelabel error page with an error. This makes it obvious that something went wrong. The error also appears in the log output. We see the Whitelabel error page because we called the endpoint from the browser. Normally, web services are invoked automatically and not manually.

Let’s go through the responses for different errors. The syntax is from the HTTP client of IntelliJ, but the calls are transferable to all other clients.

Error 1: Without query string:

```
GET http://localhost:8080/api/quotes
```

Response:

```
{
  "timestamp": "…",
  "status": 500,
  "error": "Internal Server Error",
  "path": "/api/quotes"
}
```

The status code 500 shows us an error. A timestamp is also given and the path that leads to the internal server error.

Note

If the *Spring Developer Tools* (dev tools) are included, there is more information! The following results are displayed without the dev tools.

Error 2: Wrong parameter type:

```
GET http://localhost:8080/api/quotes?index=unicorn
```

Response:

```
{
  "timestamp": "…",
  "status": 400,
  "error": "Bad Request",
  "path": "/api/quotes"
}
```

Error 3: Error due to internal `ArrayIndexOutOfBoundsException`:

```
GET http://localhost:8080/api/quotes?index=99999
```

Response:

```
{
  "timestamp": "…",
  "status": 500,
  "error": "Internal Server Error",
  "path": "/api/quotes"
}
```

9.10.3 Configuration Properties server.error.*

The details without the dev tools are reduced, but with a few configuration properties, more details can be displayed. The first four settings determine how extensive the messages in the JSON document are (see Table 9.2).

Key	Values	Additional Field
`server.error.include-binding-errors`	`always`, `on_param`, `never` (default)	`"errors"`
`server.error.include-exception`	`true`, `false` (default)	`"exception"`
`server.error.include-message`	`always`, `on_param`, never	`"message"`
`server.error.include-stacktrace`	`always`, `on_param`, `never` (default)	`"trace"`

Table 9.2 Configuration for Detailed Error Messages

The specification with the exception is a `boolean`; for the others, there are three possibilities: `never`, `always`, and `on_param` (control via query parameter). The fact that no details are displayed by default reduces security risks.

In addition, there are two further settings for the path and for whether the Whitelabel error page should appear (see Table 9.3).

Key	Default	Meaning
`server.error.path`	`/error`	Specifies the path to the error controller
`server.error.whitelabel.enabled`	`true`	Specifies whether the default error page should appear

Table 9.3 Further Settings for Detail Errors

We can increase the number of details for testing, for example, via the *application.properties* file:

```
server.error.include-message=always
server.error.include-binding-errors=always
server.error.include-stacktrace=on_param
server.error.include-exception=true
```

If we call the endpoints again, we see significantly more error details:

Error 1: Without query string:

```
GET http://localhost:8080/api/quotes
```

Response:

```
{
  "timestamp": "…",
  "status": 500,
  "error": "Internal Server Error",
  "exception": "java.lang.IllegalStateException",
  "message": "Optional int parameter 'index' is present but cannot be translated
into a null value due to being declared as a primitive type. Consider declaring
it as object wrapper for the corresponding primitive type.",
  "path": "/api/quotes"
}
```

Error 2: Wrong parameter type:

```
GET http://localhost:8080/api/quotes?index=unicorn
```

Response:

```
{
  "timestamp": "…",
  "status": 400,
  "error": "Bad Request",
  "exception": "org.springframework.web.method.annotation.
MethodArgumentTypeMismatchException",
  "message": "Failed to convert value of type 'java.lang.String' to required
type 'int'; For input string: \"unicorn\"",
  "path": "/api/quotes"
}
```

Error 3: Error due to internal `ArrayIndexOutOfBoundsException`:

```
GET http://localhost:8080/api/quotes?index=1111111
```

Response:

```
{
  "timestamp": "…",
  "status": 500,
  "error": "Internal Server Error",
  "exception": "java.lang.ArrayIndexOutOfBoundsException",
  "message": "Index 1111111 out of bounds for length 3",
  "path": "/api/quotes"
}
```

If the `server.error.include-stacktrace=on_param`, the JSON response can include the stack trace if the query parameter `trace=true` is specified. Here's an example:

```
GET http://localhost:8080/api/quotes?index=1111111&trace=true
```

Response:

```
{
  "timestamp": "…",
  "status": 500,
  "error": "Internal Server Error",
  "exception": "java.lang.ArrayIndexOutOfBoundsException",
  "trace": "java.lang.ArrayIndexOutOfBoundsException: Index 1111111 out of
bounds for length 3\r\n\tat com.tutego.boot.rest.QuoteRestController.
retrieveQuote(QuoteRestController.java:19)\r\n\tat […]java.base/
java.lang.Thread.run(Thread.java:833)\r\n",
  "message": "Index 1111111 out of bounds for length 3",
  "path": "/api/quotes"
}
```

In production applications, `include-stacktrace` should remain on `never` to avoid exposing internal details to the outside world. The same exception appears in the log output.

9.10.4 Exception Resolver

The automatic translation of exceptions into corresponding status codes is handled by an exception resolver in Spring Framework. There is a built-in exception resolver in Spring Web MVC that translates predefined errors from the container to HTTP status codes.[39] This is meant for "hard" errors from the 4** and 5** range.

Much more interesting is the question of how to map custom exceptions that come from business logic, for example, to corresponding status codes and error messages.

9.10.5 Map Exception to Status Code with @ResponseStatus

We have the option to code our own exception class and tag it with `@ResponseStatus` so that when the exception arrives at Spring Web MVC, it can be translated into a status code and message. Let's see an example of this. The annotation on an exception class can specify an HTTP status code and a reason:

```
@ResponseStatus( value  = HttpStatus.NOT_FOUND,
                 reason = "No such quote" )
class QuoteNotFoundException extends RuntimeException { }
```

The `QuoteNotFoundException` class is a normal `RuntimeException` and annotated with `@ResponseStatus`.[40] Two annotation attributes are set: `value` specifies the HTTP status code, and `reason` is a brief description that can't be changed dynamically.

The controller could throw the exception. This is done here only as an example because normally, there would be no `try-catch` block in the controller to rethrow the exception:

```
@GetMapping
public String retrieveQuote( int index ) {
try { return QUOTES[ index ]; }
catch ( Exception e ) { throw new QuoteNotFoundException(); }
}
```

A call in the HTTP client with `GET  http://localhost:8080/api/quotes?index=111111` would return the following message:

```
{
  "timestamp": "…",
  "status": 404,
  "error": "Not Found",
  "exception": "com.tutego.boot.rest.QuoteNotFoundException",
```

---

[39] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/servlet/mvc/support/DefaultHandlerExceptionResolver.html*

[40] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/bind/annotation/ResponseStatus.html*

```
  "message": "No such quote",
  "path": "/api/quotes"
}
```

@ResponseStatus: Evaluation

The `@ResponseStatus` annotation poses a problem as it's not suitable for an exception class. If the business logic throws an exception, it goes against the onion architecture's principles. Additionally, the `@ResponseStatus` annotation's name indicates that it's solely for the status code. This would be acceptable if the exception classes were declared outside of the controllers.

Another issue with `@ResponseStatus` is that it doesn't allow for customization of error messages. For instance, it doesn't permit expressing the range of values for the index. However, there is an alternative solution available for this problem.

9.10.6 ResponseStatusException

Within the controller, you can throw a special `ResponseStatusException`,[41] which is a class provided by Spring Web MVC. This exception provides multiple constructors, so we can pass a status code, a reason, and even a `Throwable`:

- `ResponseStatusException(HttpStatus status)`
- `ResponseStatusException(HttpStatus status, String reason)`
- `ResponseStatusException(HttpStatus status, String reason, Throwable cause)`
- `ResponseStatusException(int rawStatusCode, String reason, Throwable cause)`

While `@ResponseStatus` is declarative, `ResponseStatusException` can be used to programmatically wrap the status code, a description, and a `Throwable`. All three specifications can be determined flexibly at runtime. Here's an example:

```
@GetMapping
public String retrieveQuote( int index ) {
  try {
    return QUOTES[ index ];
  }
  catch ( Exception e ) {
    throw new ResponseStatusException( HttpStatus.NOT_FOUND,
                                       "No such quote",
                                       e );
  }
}
```

---

[41] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/server/ResponseStatusException.html*

If we send `GET  http://localhost:8080/api/quotes?index=111111` via the HTTP client, then the response is as follows:

```
{
  "timestamp": "…",
  "status": 404,
  "error": "Not Found",
  "exception": "org.springframework.web.server.ResponseStatusException",
  "message": "No such quote",
  "path": "/api/quotes"
}
```

If we use `ResponseStatusException` on the controller level, this saves exception classes. A separate subclass is also possible:

```
class QuoteNotFoundException extends ResponseStatusException {
  public QuoteNotFoundException( String reason ) {
    super( HttpStatus.NOT_FOUND, reason );
  }
}
```

The handler method indirectly throws `ResponseStatusException`:

```
@GetMapping
public String retrieveQuote( int index ) {
  try {
    return QUOTES[ index ];
  }
  catch ( Exception e ) {
    throw new QuoteNotFoundException(
                "No such quote with index " + index );
  }
}
```

The JSON result would be slightly different because it shows our `QuoteNotFoundException`:

```
{
  "timestamp": "2022-10-03T12:34:33.364+00:00",
  "status": 404,
  "error": "Not Found",
  "exception": "….QuoteNotFoundException",
  "message": "No such quote with index 111111",
  "path": "/api/quotes"
}
```

A `ResponseStatusException` has no place on the business logic side. Although we could query an exception in the controller and throw a `ResponseStatusException`, this isn't a complete solution.

9.10.7 Local Controller Exception Handling with @ExceptionHandler

So far, we've thrown an exception from the controller. This was either done unintentionally, as with the `ArrayIndexOutOfBoundsException`, or we deliberately used `@ResponseStatus` to generate an exception with a status code. Alternatively, we could use the specialized Spring Web MVC `ResponseStatusException`, which allows us to set both a status code and a message.

There is an alternative method to automatically convert exceptions into a status code and an error message. The approach can be divided into two categories: local "translation" specific to the controller or global "translation" that applies to all controllers.

In the local variant, we can include methods in a controller to handle exceptions thrown during request processing within that controller. These methods are annotated with `@ExceptionHandler`. Here's an example:

```
@RestController
@RequestMapping( "/api/quotes" )
public class QuoteRestController {

  …

  @ResponseStatus( value  = HttpStatus.NOT_FOUND,
                   reason = "No such quote" )
  @ExceptionHandler( IndexOutOfBoundsException.class )
  public void noSuchQuote() { }

  @GetMapping
  public String retrieveQuote( int index ) {
    return QUOTES[ index ];
  }
}
```

An attribute in `@ExceptionHandler`[42] determines which exception should be caught and translated. The `@ResponseStatus` annotation, which we've already seen, appears again. What's new is that `@ExceptionHandler` encodes the mapping of exceptions directly into the controller. The "artificial" method serves the declarative mapping of exceptions to the status code. The exception is no longer associated with a `ResponseStatus`.

---

[42] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/bind/annotation/ExceptionHandler.html*

If you call `GET http://localhost:8080/api/quotes?index=111111` in the HTTP client, the result is as follows:

```
{
  "timestamp": "…",
  "status": 404,
  "error": "Not Found",
  "exception": "java.lang.ArrayIndexOutOfBoundsException",
  "message": "No such quote",
  "path": "/api/quotes"
}
```

Catching Exceptions with the @ExceptionHandler Method

In our example, the `@ExceptionHandler` method has no parameters and no return:

```
@ResponseStatus( value  = HttpStatus.NOT_FOUND,
                 reason = "No such quote" )
@ExceptionHandler( IndexOutOfBoundsException.class )
public void noSuchQuote() { }
```

It's possible to make changes in this regard. By using `@ExceptionHandler`, the information about which exceptions to catch can be omitted, and the exception type can be added to the parameter list:

```
@ResponseStatus( value  = HttpStatus.NOT_FOUND,
                 reason = "No such quote" )
@ExceptionHandler
public void noSuchQuote( IndexOutOfBoundsException e ) { }
```

If you try it this way, you'll get the same information as before:

```
{
  "timestamp": "…", "status": 404, "error": "Not Found",
  "exception": "java.lang.ArrayIndexOutOfBoundsException",
  "message": "No such quote", "path": "/api/quotes"
}
```

The advantage is that the method can evaluate the exact information from this exception.

@ExceptionHandler with ResponseEntity

The `@ExceptionHandler` method can return a `ResponseEntity` object. This allows the method to build and pass up the status code, header, and body individually. Here's an example:

```
@ExceptionHandler
public ResponseEntity<String> indexOutOfBounds( IndexOutOfBoundsException e ) {
  return ResponseEntity.status( HttpStatus.NOT_FOUND )
                       .body( e.getMessage() );
}
```

If you call `GET http://localhost:8080/api/quotes?index=111111` in the HTTP client, you get the following result:

```
HTTP/1.1 404
Content-Type: application/json
Content-Length: 39
Date: Mon, 03 Oct 2022 16:11:02 GMT
Keep-Alive: timeout=60
Connection: keep-alive

Index 111111 out of bounds for length 3
```

With this approach, we have complete control over the body of the response. However, it's not appropriate for the content type to be JSON when the body contains plain text.

9.10.8 RFC 7807: "Problem Details for HTTP APIs"

Status codes are often imprecise and don't convey details about what went wrong. For instance, a status code of 400 for **Bad Request** is too general and doesn't provide enough information to the client. However, the server can send a more comprehensive error object instead of a simple string. While Spring generates a default error object in case of an error, we can also specify our own custom error object.

To avoid different formats for the error specification, a standard has been documented in *RFC 7807: Problem Details for HTTP APIs* (*www.rfc-editor.org/rfc/rfc7807*). The standard describes fields that are usually returned as JSON documents in case of an error.

The HTTP status code remains, and the details in the JSON document are as follows:

- **`instance` (URI string)**
  The resource with the issue.
- **`status` (number)**
  The HTTP status code.
- **`type` (URI string)**
  The problem type.
- **`title` (string)**
  A summary of the problem.
- **`detail` (string)**
  A readable longer description of the problem.

The fields are optional, and more fields can be added.

ProblemDetail

Spring programs can also send detail information according to this standard because there is a data type in the library called `ProblemDetail`.[43] The code looks like this:

```
@ExceptionHandler
public ResponseEntity<?> indexOutOfBounds( IndexOutOfBoundsException e ) {
  var problemDetail = ProblemDetail.forStatus( HttpStatus.NOT_FOUND );
  problemDetail.setTitle( "The index is out of bound" );
  problemDetail.setDetail( e.getMessage() );
  problemDetail.setType( URI.create(
        "https://docs.oracle.com/en/java/javase/17/docs/api/java.base/" +
        "java/lang/IndexOutOfBoundsException.html" ) );
  return ResponseEntity.of( problemDetail ).build();
}
```

As you know, we have an `@ExceptionHandler` that responds to an `IndexOutOfBoundsException` (the base type of `ArrayIndexOutOfBoundsException`) and returns a `ResponseEntity` object. The static `forStatus(…)` method builds a `ProblemDetail` object, and the setters set the title, details, and error type. But the `@ExceptionHandler` methods can also return these `ProblemDetails` directly.

This is the answer:

```
{
  "type": "https://docs.oracle.com/en/…/IndexOutOfBoundsException.html",
  "title": "The index is out of bound",
  "status": 404,
  "detail": "Index 111111 out of bounds for length 3",
  "instance": "/api/quotes"
}
```

For the type of error, refer to the Java documentation.

Task: Start the program with the dev tools or enable more details for the errors using the following settings:

```
server.error.include-message=always
server.error.include-binding-errors=always
server.error.include-stacktrace=on_param
server.error.include-exception=true
```

The wrong call to our endpoint *http://localhost:8080/api/stat/last-seen?start=2022-01&end=20-23-05* will result in an error of the following type:

---

[43] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/http/ProblemDetail.html*

```
{
  "timestamp": "…",
  "status": 400,
  "error": "Bad Request",
  "exception":
"org.springframework.web.method.annotation.MethodArgumentTypeMismatchException",
  "message": "Failed to convert value of type 'java.lang.String' to required
type 'java.util.Optional'; Failed to convert from type [java.lang.String] to
type [@org.springframework.web.bind.annotation.RequestParam java.time.YearMonth]
for value [20-23-05]",
  "path": "/api/stat/last-seen"
}
```

This time, the exception isn't coming from inside the controller, but from Spring Web MVC. Develop an `@ExceptionHandler` that builds and returns a meaningful `ProblemDetail`.

Proposed solution: A new method is annotated with `@ExceptionHandler` and placed in the controller. The method's parameter is of type `MethodArgumentTypeMismatchException`, which gives us access to the erroneous parameter name (`e.getName()`) and to the erroneous value (`e.getValue()`):

```
@ExceptionHandler
public ResponseEntity<?> yearMonthMisformatted(
                            MethodArgumentTypeMismatchException e ) {
   var problem = ProblemDetail.forStatus( HttpStatus.BAD_REQUEST );
   problem.setTitle( "The format of the argument '" + e.getName() + "' is wrong"
 );
   problem.setDetail( "The format of '" + e.getValue() + "' is invalid. The form
at string must match ISO 8601 format 'YYYY-MM', like '2025-03'" );
   problem.setType( URI.create(
           "https://www.iso.org/iso-8601-date-and-time-format.html" ) );
   return ResponseEntity.of( problem ).build();
}
```

9.10.9 Global Controller Exception Handling with Controller Advice

Until now, we've placed the `@ExceptionHandler` locally in the respective controller class. However, an `IndexOutOfBoundsException` can also occur in other controllers, making it useful to define an exception handler globally. This helps to avoid duplicating `@ExceptionHandler` implementations in different controllers.

Global exception handlers are programmed via a *controller advice*. The well-known `@ExceptionHandler` can be put into its own class, not into a controller, but into a new class annotated with `@ControllerAdvice`.[44] This allows the local `@ExceptionHandler` to be pulled out and defined globally for all controllers:

```
@ControllerAdvice
class IndexOutOfBoundsControllerAdvice {

  @ExceptionHandler
  public ProblemDetail indexOutOfBounds( IndexOutOfBoundsException e ) {
    return ProblemDetail.forStatusAndDetail( HttpStatus.NOT_FOUND,
                                             e.getMessage() );
  }

}
```

Multiple `@ExceptionHandler` methods can also be placed in the class, which can then react to different exception objects.

Control Application from @ControllerAdvice

Sometimes, `@ControllerAdvice` shouldn't be global and shouldn't apply the same error mapping for every controller. In such cases, we may want something that is specific to certain controllers but not necessarily to every controller. To achieve this, we can make use of the `@ControllerAdvice` annotation:

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Component
public @interface ControllerAdvice {
 @AliasFor("basePackages") String[] value() default {};
 @AliasFor("value") String[] basePackages() default {};
 Class<?>[] basePackageClasses() default {};
 Class<?>[] assignableTypes() default {};
 Class<? extends Annotation>[] annotations() default {};
}
```

As you can see, there's a normal `@Component` and annotation attributes that look familiar to us because at least `basePackages*` matches attributes from `@ComponentScan` in name. This can be used, for example, to determine which packages these `@ControllerAdvice` classes should apply to.

Here are two examples:

```
@ControllerAdvice( "com.tutego.date4.interfaces.rest.profile" )
@ControllerAdvice( NANisError.class )
```

---

[44] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/bind/annotation/ControllerAdvice.html*

The first usage of `@ControllerAdvice` specifies that it only applies to controllers in a particular package. Similarly, in the second usage, it only applies to controllers annotated with `NANisError`.

9.11 RESTful API

A *web service* is a software system that enables communication and interaction between different applications over the internet. Typically, web services operate through an HTTP interface, which allows remote clients to call upon the functionality of the service.

Two standards have emerged as the most popular approaches to web service implementation: SOAP-based web services and REST-based web services, also known as RESTful APIs.

- **SOAP-based web services**
  These web services use the Simple Object Access Protocol (SOAP) to exchange structured data between clients and servers. SOAP messages are typically sent over HTTP and are designed to be platform- and language-independent.
- **REST-based web services**
  In contrast to SOAP-based web services, these web services use the Representational State Transfer (REST) architectural style to implement a lightweight and flexible approach to web service development. RESTful APIs use HTTP methods such as `GET`, `POST`, `PUT`, and `DELETE` to perform operations on resources identified by unique URLs.

Today, when people refer to "web services," the term RESTful API has become increasingly synonymous with the concept. This is due in part to the popularity of REST-based web services, which have gained widespread adoption thanks to their simplicity, flexibility, and ease of use.

9.11.1 Principles behind REST

To ensure that a web service can truly be considered RESTful, it must adhere to several core principles that underlie the REST architecture. First, resources within a RESTful web service must be uniquely identified by a URL. To access a particular resource, clients must use the appropriate HTTP method—such as `GET` to read a resource, `POST` to create a new resource, `PUT` to update an existing resource, or `DELETE` to remove a resource. These HTTP methods remind us of the CRUD operations (create, read, update, delete) that we perform on databases.

Another key principle of RESTful web services is that client-server communication must always be stateless. This means that after a client interacts with the server, the server forgets all about the client and doesn't store any state information. This approach is particularly useful for building distributed applications, as it allows clients to move seamlessly between different machines on the network without disrupting the operation.

To ensure good performance in a stateless architecture, data should be sourced from a database or other reliable storage mechanism, and a distributed cache can help to improve performance even further. By adhering to these core principles, a web service can be considered truly RESTful and able to provide the simplicity, scalability, and flexibility that are hallmarks of this approach.

An API that works according to these RESTful principles is called a *RESTful API*. There are other criteria, such as the *Richardson Maturity Model*. It defines three levels; our implementation moves to level 2.

Note: REST

The term *REST* was first introduced by Roy Fielding in his dissertation titled “Architectural Styles and the Design of Network-Based Software Architectures.” Fielding is also renowned for coauthoring the HTTP specification and being a cofounder of the Apache Software Foundation. At its core, REST is a style of software architecture that defines a set of constraints for building web services. One of the key principles of REST is that when a client and server communicate, they transfer a representation of the state of a resource. This representation can take the form of various data formats, such as JSON or XML, depending on the needs of the application.

What is important in a RESTful web service is that the resource itself remains the same, regardless of the format in which it’s exchanged. For example, if a client requests a specific resource from a server, the server will send back a representation of that resource in the requested format. The client can then use this representation to interact with the resource, updating its state or retrieving additional information as needed.

REST in all its “glory”—as Martin Flower calls it[45]—supports self-description of endpoints. This means that an endpoint refers to other endpoints and also tells you what you can do with those endpoints. In this context, the term *Hypermedia as the engine of application state* (HATEOAS) is often used. Although it’s a nice idea and *Spring HATEOAS* (*https://spring.io/projects/spring-hateoas*) supports the construction of linked documents, it hardly plays a role in practice.

Path Variable versus Query Parameter

In RESTful web services, resources are identified using a URI, which serves as a unique identifier for the resource. In a RESTful API, the URI for a resource shouldn’t contain any query parameters or fragments. However, this doesn’t mean that query parameters are completely forbidden. In fact, they can be used to provide metadata about the resource, such as its format, sorting, or pagination.

---

[45] *https://martinfowler.com/articles/richardsonMaturityModel.html*

It's worth noting that query parameters in a RESTful API shouldn't modify the resource itself, but only its presentation. In other words, the resource remains the same regardless of the query parameters used to access it. This approach ensures that RESTful web services adhere to the principle of resource identification and representation, which is one of the fundamental tenets of the REST architectural style.

Idempotent and Safe

In the context of RESTful web services, it's important for a server to be stateless. This means that the server shouldn't store any information about the client between multiple HTTP requests. However, this doesn't mean that the application can't use caching to improve performance. Caching is an essential technique that can significantly enhance the responsiveness of an application.

Note

The term "stateless" doesn't imply that a web service can't perform write operations. In fact, it's perfectly acceptable for a web service to perform write operations. However, the key point is that the server shouldn't maintain any state information about the client between HTTP requests.

When it comes to implementing RESTful web services, there are two crucial concepts that we must consider, especially with operations: *idempotency* and *safety*. It's important to ensure that certain operations are idempotent and some are safe because a client may send a request multiple times due to a broken connection, and the server must respond consistently in such cases.

Idempotent requests are those that always have the same effect, regardless of the number of times they are executed. For example, the math methods `sin(…)` or a common setter are good examples of idempotent methods because they always leave the same state, no matter how many times they are called. In contrast, methods such as `random()` always return a different value and are therefore stateful.

Another important concept is safety. Safe methods don't modify resources, and they are always idempotent. This means that if you call a safe method multiple times, it won't modify the resource in any way, and the result will always be the same. In contrast, unsafe methods can't be called repeatedly without changing the resource, which would be similar to calling the `append(…)` method in a `StringBuilder` or `add(…)` in a list multiple times.

By ensuring that certain operations are idempotent and some are safe, we can also ensure that our RESTful web services are robust and reliable. This approach enables clients to execute operations without worrying about unintended side effects, and it helps to prevent data corruption and other issues that can arise when working with distributed systems.

The HTTP methods shown in Table 9.4 must be idempotent or safe.

HTTP Method	Idempotent	Safe
`OPTIONS`	Yes	Yes
`GET`	Yes	Yes
`HEAD`	Yes	Yes
`PUT`	Yes	No
`DELETE`	Yes	No
`POST`	No	No
`PATCH`	No	No

Table 9.4 HTTP Methods That Must Be Idempotent or Safe

Let's go over the core HTTP methods from Table 9.4 briefly:

- **`OPTIONS`**
  This method isn't commonly used in practice. Its purpose is to indicate which HTTP methods are permitted on an endpoint. This operation is considered idempotent, as the result remains the same regardless of how many times it's called. Additionally, this method is safe because it doesn't modify the resource.
- **`GET`**
  This method retrieves resources and can be executed multiple times with the same result, regardless of any changes to the resource in between. This operation is considered safe because it only involves reading and doesn't modify anything. Even when the server caches resources, it doesn't constitute a state change on the server in the strict sense.
- **`HEAD`**
  This method is similar to `GET`, but without the response body. Some servers may execute a full `GET` request and forward the header, content type, and content length, but discard the body. This approach can be costly. `HEAD` can be beneficial in determining the size of a resource in advance. This method is also idempotent and safe.
- **`PUT`**
  This method isn't considered safe because it can overwrite existing entities. However, if the same entity is overwritten multiple times, the result remains constant, making `PUT` idempotent.
- **`DELETE`**
  This method is considered idempotent because the server's state remains unchanged regardless of whether the same resource is deleted once or multiple

times. However, the status should be updated from **OK** to **Not Found**. This method isn't safe because it modifies the data store.

- **POST** and **PATCH**
  These methods are considered neither idempotent nor safe. `POST` creates a new entity, while a `PATCH` operation may not be idempotent because its effects aren't clearly defined. Some `PATCH` requests may append elements or increment values, causing the results to vary.

We've examined two key principles that should be followed when designing RESTful web services. The first is that a resource must be identified by a URI, and path variables are used for variable values. Additionally, there are specific requirements for HTTP methods to ensure their idempotent and safe properties.

However, it's also important in practice to use status codes correctly, which we'll discuss next.

HTTP Status Codes

There are 50 status codes, and they can be divided into five different classes:

- 1xx
  Informational responses indicating that the server has received the request and is continuing to process it. These aren't common in practice.
- 2xx
  Successful responses indicating that the request was valid and the operation was executed or initiated as expected. The most common status code in this category is 200 (OK).
- 3xx
  Redirection responses indicating that the requested resource has been moved to a new location or is available in the cache.
- 4xx
  Client error responses indicating that the request was invalid or could not be processed due to a client-side error, such as a missing parameter. The most common status code in this category is 400 (**Bad Request**).
- 5xx
  Server error responses indicating that the server was unable to fulfill the request due to an internal error, such as a database connection failure. The most common status code in this category is 500 (**Internal Server Error**).

If a server uses highly specific status codes, it can create an additional burden for the client as the client must also handle each individual HTTP status code in the response. As a result, it's generally recommended to use more general status codes instead of overly specific ones.

Which status codes now play a role in practice? Table 9.5 lists the most important ones. The numeric HTTP status code is listed in the first column, while the second column contains the stated reason phrase for the error according to the specification.

Code	Reason Phrase	Explanation
200	**OK**	The request was carried out successfully. The result is included in the answer.
201	**Created**	The request performed successfully. The resource was created.
204	**No Content**	The request performed successfully. The response doesn't contain any data.
304	**Not Modified**	The content of the requested resource is unchanged since the last query. The response contains only headers, but no body because that can be read from the cache.
400	**Bad Request**	The request was built incorrectly.
401	**Unauthorized**	The request can't be performed without valid authentication.
403	**Forbidden**	The client has no authorization.
404	**Not Found**	The resource wasn't found.
409	**Conflict**	There was a constraint violation.
500	**Internal Server Error**	An unexpected server error occurred.

Table 9.5 Status Codes Relevant to Practice

Common Status Codes for HTTP Methods

With the different HTTP methods, the three status codes in Table 9.6 appear again and again.

HTTP Method	Code
`GET`	200 (**OK**)
`POST`	201 (**Created**)
`PUT`	200 (**OK**)

Table 9.6 Common Status Codes for Known HTTP Methods

When a requested resource can't be found, it's appropriate to return a 404 status code. There are other HTTP status codes that a server could provide:

- **202 (Accepted)**
  Some servers use this status code for certain operations to indicate that the request has been accepted but isn't yet complete. This allows the server to quickly respond and avoid timeouts for long requests. However, it can be risky for clients to always consider a 202 response as successful because the request may still fail in the background.
- **204 (No Content)**
  Some servers use this status code instead of a 200 for certain operations, such as a `DELETE` request, when there is no result and the response body is empty. This is a valid use of the status code.

9.11.2 Implement REST Endpoints for Profiles

After establishing the theoretical foundation for RESTful APIs, we aim to create a REST endpoint for profiles within the Date4u application. Our goal is to implement the complete range of CRUD operations shown in Table 9.7.

HTTP Method	Path Fragment	Result
`GET`	`/api/profiles`	Returns all profiles as a JSON array.
`POST`	`/api/profiles`	Accepts and saves a JSON document for a new profile.
`GET`	`/api/profiles/1`	Returns the profile with ID 1 as the JSON return.
`PUT`	`/api/profiles/1`	Accepts a JSON object and updates the profile with ID 1.
`DELETE`	`/api/profiles/1`	Deletes the profile with ID 1.

Table 9.7 Endpoints for RESTful Web Services

Usually, endpoints are always named in the plural, that's why it's called `profiles` and not `profile`. In databases, table names are usually singular.

Note: Singleton versus Collection

A single element returned from an API is referred to as a *singleton*, while a group of elements is referred to as a *collection*. A `GET` request to `/api/profiles` will return a collection, but a `GET` request on `/api/profiles/1` will return a singleton.

Framework for a RESTful Web Service

We've already started to build a `ProfileRestController` with the following central parts:

```
@RestController
@RequestMapping( "/api/profiles" )
public class ProfileRestController {

  @Autowired ProfileRepository profiles;

}
```

Task: Implement the GET Access for a Collection

Implement `GET` access to `/api/profiles` so that all profiles are returned.

The solution is supposedly simple:

```
@GetMapping
public List<Profile> profiles() {
  return profiles.findAll();
}
```

A `GET` request on *http://localhost:8080/api/profiles* should return a list of JSON documents. However, an exception message with the error **Premature end of chunk coded message body** is thrown. This is because the Jackson JSON library attempts to convert the entire `Profile` object and all associated objects, such as `Unicorn`, `Photo`, and `Profile` (via the `Likes` table), to JSON. Due to bidirectional relationships, this process can lead back to the starting point, resulting in an endless loop that ultimately causes a stack overflow error.

There are different solutions to the problem:

- **Ignoring the referenced objects via the Jackson annotation `@JsonIgnore`**[46]
  Let's set the following annotations in `Profile`:

  ```
  class Profile {
    …
    @JsonIgnore
    @OneToOne( … )
    private Unicorn unicorn;

    @JsonIgnore
    @OneToMany( … )
    private List<Photo> photos = new ArrayList<>();
    …
  }
  ```

---

[46] *www.javadoc.io/doc/com.fasterxml.jackson.core/jackson-annotations/latest/com/fasterxml/jackson/annotation/JsonIgnore.html*

To solve this issue, we can modify the endpoint to return an array of profile objects with only the basic attributes such as `id`, `nickname`, `birthdate`, `manelength`, `attractedToGender`, `description`, and `lastseen`. The associated elements such as `Unicorn` and `Photo` are excluded from the response.

- **Hiding the back references**

  This is another solution that is also Jackson-specific in which we hide the back references using the `@JsonBackReference`[47] annotation. This will hide the route back that causes the circular reference issue. For example, you can use the `@JsonBackReference` annotation in the `Photo` class as follows:

  ```
  class Photo {
    …
    @ManyToOne @JoinColumn(name = "profile_fk")
    @JsonBackReference
    private Profile profile;
    …
  }
  ```

  The reference to the object can be annotated in `Profile` with `@JsonManagedReference`[48] (for the forward reference).

Both solutions are suboptimal because they involve sending information directly from the entity bean to the client. When a request is made, the controller accesses the repository, which uses *Jakarta Persistence* to retrieve entity beans that are automatically converted to JSON. To address circular reference issues, we added a Jackson-specific annotation to our entity bean. However, this approach is fundamentally flawed because the entity bean and the controller are on different levels and shouldn't be responsible for JSON representation. To resolve this issue, we can use specialized transfer objects that don't include referenced objects.

9.11.3 Data Transfer Objects

Entity beans are designed to serve as containers, but they shouldn't be directly sent out to clients as JSON using Jackson. This is because clients may require different representations or projections, and it's necessary to separate the concerns of data transfer and entity representation. To address this, *data transfer objects* (DTOs) are commonly used. These special containers are designed solely for transferring data between different layers of an application.

---

[47] *www.javadoc.io/doc/com.fasterxml.jackson.core/jackson-annotations/latest/com/fasterxml/jackson/annotation/JsonBackReference.html*

[48] *www.javadoc.io/doc/com.fasterxml.jackson.core/jackson-annotations/latest/com/fasterxml/jackson/annotation/JsonManagedReference.html*

When a controller receives an entity bean from a service or repository, instead of passing on the entity bean, it should transfer the data from the entity bean to a DTO. DTOs are simple container classes that can be either records or classes with `public` variables. This way, the DTO contains only the required data for the client, without the need to expose the entire entity bean.

ProfileDto

A record might look like this:

```
public record ProfileDto(
  Long id, String nickname, LocalDate birthdate, int manelength,
  int gender, Integer attractedToGender, String description,
  LocalDateTime lastseen
) { }
```

Listing 9.14 ProfileDto.java

The purpose of the DTO is to hold only the relevant data for a particular scenario, excluding unnecessary details such as the unicorn, photos, and data from the `Likes` table. Other endpoints may provide information about the likes. The DTO can be placed in the same package as the REST class because it's only visible to the controller classes.

An instance method should help in converting a `Profile` object to a `ProfileDto` object:

```
class ProfileMapper {
  ProfileDto convert( Profile p ) {
    return new ProfileDto(
        p.getId(), p.getNickname(), p.getBirthdate(), p.getManelength(),
        p.getGender(), p.getAttractedToGender(), p.getDescription(),
        p.getLastseen()
    );
  }
}
```

Listing 9.15 ProfileMapper.java

In the `ProfileRestController`, we can create a new instance variable and later initiate the conversion to a DTO in the handler methods:

```
public class ProfileRestController {
  private final ProfileMapper profileMapper = new ProfileMapper();
…
}
```

Listing 9.16 ProfileRestController.java Extension

To summarize, the big advantage of working with DTOs is that we can evolve and change the data types independently of the entity data types. If something changes later in the entity bean, we don't necessarily have to make that change in the DTO, we just have to rewrite the mapping a bit. So, let's do that next.

Map a Data Transfer Object

To enable data exchange between the entity bean and the DTO, the controller must perform mapping in both directions in these specific ways:

- When the controller requests data from the service/repository, the retrieved entity bean data must be mapped to the corresponding DTO.
- Conversely, when the client sends data, the controller must copy it from the DTO container to the entity bean.

While manual mapping is an option, Section 9.16.3 will explore automatic mapping techniques.

Task: Who Is Allowed to See Whom?

The `Profile` class belongs in the `*.core.*` package. In which packages should `ProfileDto` and `ProfileMapper` belong?

Proposed solution: If you base this on the onion architecture, `ProfileDto` and `ProfileMapper` wouldn't be correctly placed in the core. Controllers lie outside of the core and are allowed to access it. That means the controller can see the core's entities, but under no circumstances should an entity look outward and see data types of the controller—and the DTO lies outside, as it's only a tool for the controller, just like the mapper. In addition, different controllers might want to use different transfer objects from the same entity bean `Profile`.

Task: Implement GET with a Data Transfer Object Return

Given the understanding that transfer objects should be used, the task at hand is to modify the endpoint's handler methods. Specifically, the goal is to revise the `GET` method for the `/api/profiles` path so that it returns a collection of DTOs.

Proposed solution: After `findAll()` has given us a list of `Profile` objects, we can build up a stream with `stream()` and then map it to the DTOs with `map(…)`:

```
@GetMapping
public List<ProfileDto> profiles() {
  return profiles.findAll().stream()
                           .map( profileMapper::convert ).toList();
}
```

Listing 9.17 ProfileRestController.java Extension

A handler method can provide a `Stream` directly, so the code can be abbreviated. There are other solutions as well. We could work with Spring Data projections, which Chapter 7, Section 7.12, shows in detail. However, this only works in the direction toward the client.

9.11.4 Best Practice: Don't Deliver a Complete Collection at Once

The following solution is compact, but there is a weak point:

```
@GetMapping
public Stream<ProfileDto> profiles() {
  return profiles.findAll().stream().map( profileMapper::convert );
}
```

We don't really want to deliver a complete collection when there might be millions of entries in the database. With an endpoint, you'll generally never return all entities, but always use pagination. We won't program this pagination manually because Spring provides a technique for doing so. We consider the subject again in Section 9.13.2. Currently, the return isn't a problem with our 20 profiles in the database.

9.11.5 GET and DELETE on Individual Resources

Let's move on to two additional methods. A client should also be able to specifically request a profile.

Task: GET and DELETE for Profiles

To load and delete a profile with a given ID, we implement methods for the following:

- `GET /api/profiles/{id}`
- `DELETE /api/profiles/{id}`

Make sure that nonexistent resources must return a 404!

Note

The handler method for `DELETE` doesn't have to be implemented; a log output is sufficient.

Proposed solution: We call the first method `get(…)`, which can look like this:

```
@GetMapping( "/{id}" )
public ResponseEntity<?> get( @PathVariable long id ) {
  Optional<ProfileDto> maybeProfileDto =
      profiles.findById( id ).map( profileMapper::convert );

  return ResponseEntity.of( maybeProfileDto );
}
```

Listing 9.18 ProfileRestController.java Extension

For the return, we have to consider the special case that there is no profile for the ID. The `ResponseEntity.of(…)` method is very useful here because it returns status code 200 and a body if `Optional` isn't empty. Otherwise, it returns 404 and `null` in the body.

The handler method for `DELETE` can look like this:

```
@DeleteMapping( "/{id}" )
public ResponseEntity<?> delete( @PathVariable long id ) {
  if ( ! profiles.existsById( id ) )
    return new ResponseEntity<>( NOT_FOUND );

  profiles.deleteById( id );
  return new ResponseEntity<>( OK );
}
```

Listing 9.19 ProfileRestController.java Extension

`DELETE` must also check if the profile with the ID exists—if not, the REST endpoint should return 404. If the deletion succeeded, we can return `OK` or `NO_CONTENT`: both status codes are okay.

Tip

With a new repository method such as `int deleteProfileById(long id)`, the return type `int` can express via `0` or `1` (number of deleted rows) whether the deletion was successful. We discussed this previously in Chapter 7, Section 7.8.2.

9.11.6 POST and PUT with @RequestBody

If we define CRUD operations in a RESTful API, then we have four operations. We've already implemented two of them: `GET` and `DELETE`. What we're missing are two operations that allow the client to create, overwrite, and update entities. For this, we use the HTTP methods `POST` and `PUT`:

- `POST /api/profiles` (create new)
- `PUT /api/profiles/{id}` (update)

The updated requirement is that the client sends a JSON object within the request body, which the handler method needs to receive.

ResponseBody and RequestBody

When the `@ResponseBody` annotation is used in Spring, it will automatically convert any nonstring, `byte[]`, or `Resource` to JSON. This applies to any return within a `@RestController`. Conversely, if the client sends data to the endpoint, the parameter variable that receives the object must be annotated with `@RequestBody`.

Here are two examples of parameter lists:

- `( @RequestBody ProfileDto profile )`
- `( @PathVariable long id, @RequestBody ProfileDto profile )`

The `@RequestBody` annotation instructs Spring Web MVC to convert the JSON object from the client into a Java object and pass it to the handler method. Similar to other annotations, it's possible to retrieve multiple values from a single HTTP message. Therefore, `@RequestBody` can be easily combined with other passed values, such as path variables or query parameters.

Task: POST for Profiles

Implement a way to upload a new profile via `POST`. The call in the HTTP client may look like this:

```
POST http://localhost:8080/api/profiles
Content-Type: application/json

{
  "nickname": "CrazyEyes",
  "birthdate": "1980-08-16",
  "manelength": 3,
  "gender": 1,
  "attractedToGender": 2,
  "lastseen": "2022-07-03T22:57:55"
}
```

Proposed solution: The handler method can be implemented as follows:

```
@PostMapping
public ResponseEntity<?> create( @RequestBody ProfileDto dto ) {

  if ( nonNull( dto.id() ) )
    return ResponseEntity.badRequest()
        .body( "ID of profile must be null but was '" + dto.id() + "'" );

  Profile newProfile = new Profile( dto.nickname(), dto.birthdate(),
                                    dto.manelength(), dto.gender(),
                                    dto.attractedToGender(),
                                    dto.description(), dto.lastseen() );
  newProfile = profiles.save( newProfile );

  return ResponseEntity
          .created( URI.create( "/api/profiles/" + newProfile.getId() ) )
          .body( profileMapper.convert( newProfile ) );
}
```

Listing 9.20 ProfileRestController.java Extension

The solution is divided into three sections:

1. **Consistency test**
   Because the database assigns the IDs, the sent profile must not have its own ID. If the ID isn't `null`, the web service aborts the operation with a `BAD REQUEST`.
2. **Mapping**
   The DTO is mapped to the entity bean and stored.
3. **Structure of the return**
   The return contains the stored object, converted back into a DTO. In addition, a location header is set.

Applied to the example task, the result from the HTTP response might look like this:

```
HTTP/1.1 201
Location: /api/profiles/33
Content-Type: application/json
Transfer-Encoding: chunked
Date: Tue, 18 Oct 2022 17:28:04 GMT
Keep-Alive: timeout=60
Connection: keep-alive

{
  "id": 33,
  "nickname": "CrazyEyes",
  "birthdate": "1980-08-16",
  "manelength": 3,
  "gender": 1,
  "attractedToGender": 2,
  "description": null,
  "lastseen": "2022-07-03T22:57:55"
}
```

Task: PUT for Profiles

Implement a handler method that can update a profile via `PUT`.

Proposed solution: The implementation isn't as short as you may have expected because, again, some errors have to be handled in this solution.

```
@PutMapping( "/{id}" )
public ResponseEntity<?> update( @PathVariable long id,
                                 @RequestBody  ProfileDto dto ) {
  if ( dto.id() != null && dto.id() != id )
    return new ResponseEntity<>(
        "Unable to update profile, because ID of path variable "
      + "does not match ID of profile", CONFLICT );
  // dto.id() could be null but we use the id from the parameter
  var maybeProfile = profiles.findById( id );
  if ( maybeProfile.isEmpty() )
    return new ResponseEntity<>(
        "Unable to update profile, because no profile with '"
      + id + "' found", NOT_FOUND );

  var profile = maybeProfile.get();
  profile.setNickname( dto.nickname() );
  profile.setBirthdate( dto.birthdate() );
  profile.setManelength( dto.manelength() );
  profile.setGender( dto.gender() );
  profile.setAttractedToGender( dto.attractedToGender() );
  profile.setDescription( dto.description() );
  profile.setLastseen( dto.lastseen() );
  profile = profiles.save( profile );
  return ResponseEntity.ok( profileMapper.convert( profile ) );
}
```

Listing 9.21 ProfileRestController.java Extension

This solution is also composed of several parts:

1. **Consistency check**
   As a parameter, the ID could be different from the ID in the DTO. If the IDs don't match, there is a status code for `CONFLICT`.
2. **Loading the profile with the ID and existence test**
   It tries to load the profile with the ID, and if it doesn't exist, `NOT FOUND` is returned, and the web service ends.
3. **Update**
   The data from the DTO is transferred to the loaded profile.
4. **Store**
   The profile is saved.
5. **Return**
   The saved profile is converted back to a DTO and returned.

9.11.7 UriComponents

To set the location header that appeared in the code of Listing 9.20, the construct `URI.create( "/api/profiles/" + id )` appeared. Due to concatenation, this isn't nice, and, besides, the path details are already there as well. Therefore, Spring provides class `UriComponents`,[49] which facilitates the assembly of URIs via an API. This class works a bit like a `java.util.Formatter`, but for URI templates.

A `UriComponents` object can be built using a `UriComponentsBuilder`, and then the path and variables can be set. Finally, the `URI` is derived from the information. Another way is to "wish" for a `UriComponentsBuilder`[50] via a parameter variable:

```
@PostMapping
public ResponseEntity<?> create(
  @RequestBody ProfileDto dto,
  UriComponentsBuilder uriBuilder
)
```

On this `UriComponentsBuilder`, the path and variables can be set, and then a `UriComponents` object can be built that finally provides the URI:

```
URI uri = uriBuilder.path( "/api/profiles/{id}" )
                    .buildAndExpand( newProfile.getId() )
                    .toUri();
return ResponseEntity.created( uri ).body( … );
```

---

[49] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/util/UriComponents.html*

[50] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/util/UriComponentsBuilder.html*

The URI template contains our path, but Spring knows that, of course. Therefore, you may also want a `ServletUriComponentsBuilder`,[51] and then Spring can dynamically detect the path via the `HttpServletRequest` object. It looks like this:

```
@PostMapping
public ResponseEntity<?> create(
  @RequestBody ProfileDto dto,
  ServletUriComponentsBuilder uriBuilder
)
```

Later, it looks like this:

```
URI uri = uriBuilder.fromCurrentRequest()
                    .path("/{id}")
                    .buildAndExpand( newProfile.getId() )
                    .toUri();
return ResponseEntity.created( uri ).body( … );
```

9.12 Asynchronous Web Requests *

When an HTTP client calls an endpoint, it blocks the request until the handler method finishes processing it. However, in some cases, the request may take longer to complete, for example, when complex background operations are triggered such as generating a PDF report that requires data to be retrieved from various databases.

9.12.1 Long Queries Block the Worker Thread

Normal web servers accept requests in the form of threads, so for each client call, there is a *worker thread* that handles the request.

Consider a query that is artificially delayed, and let's examine the consequences of such a delay.

```
@RestController
public class SlowRestController {

  // http://localhost:8080/sync
  @RequestMapping( "/sync" )
  public ResponseEntity<?> sync() throws InterruptedException {
    log.info( "{}", Thread.currentThread() );
    TimeUnit.SECONDS.sleep( 10 );
```

---

[51] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/servlet/support/ServletUriComponentsBuilder.html*

```
    return ResponseEntity.ok( "done" );
  }
}
```

If a client calls the endpoint, `sleep(10)` causes the worker thread to stall for 10 seconds. The log output shows the worker thread:

```
… INFO 21016 --- [nio-8080-exec-2] c.t.b.r.SlowRestController : ↩
Thread[http-nio-8080-exec-2,5,main]
```

When handling long-running operations, it's important to consider that most systems limit the number of worker threads to prevent overloading the system with too many open connections. In such cases, if a worker thread becomes blocked, the server must allocate a new thread to process the request, and there is a limit to the number of threads that can be allocated. When this limit is reached, the server will stop accepting new connections, which can cause issues for users.

The challenge with blocked threads is that the server can't determine the amount of work being performed, whether it's minimal or significant. If the controller method simply waits for the operation to complete, it may not be a significant burden on the system, and the server could theoretically handle more requests. However, it's difficult to determine when a long-running operation has reached a point where it's straining the system, and so it's important to consider strategies for managing such operations to prevent overloading the system.

9.12.2 Write Asynchronously to the Output

When a long-running operation is performed in the handler method, it can potentially block the worker threads and lead to a maximum limit being reached where the server won't accept any new connections. To avoid this, the actual operation can be outsourced to another thread so that the worker threads can be released quickly. This is done using asynchronous processing, where Spring Web MVC provides various ways to transfer the operation from the worker thread to other threads.

There are several data types that can be returned by the controller for asynchronous processing:

- `Callable`, also `CompletionStage`, `CompletableFuture` (data types from Java SE)
- `WebAsyncTask`
- `DeferredResult`

By using these types, Spring Web MVC handles the transfer of the operation to other threads, without the client noticing the internal server operation.

In addition to these options, asynchronous responses can also be achieved through HTTP streaming and Spring WebFlux.

9.12.3 A Handler Method Returns Callable

The first option is to have a handler method return a `Callable`:

```
// http://localhost:8080/callable-async
@RequestMapping( "/callable-async" )
public Callable<String> async() {
  return () -> {
    log.info( "{}", Thread.currentThread() );
    TimeUnit.SECONDS.sleep( 10 );
    return "done";
  };
}
```

Inside the `Callable` implementation is where the actual processing takes place. We're not actually invoking the `call(…)` method but instead invoking Spring when the handler method returns a `Callable`. This allows Spring to select a new thread to process the `Callable` and free up the worker thread immediately. Once the `call()` method has completed its execution and returns to Spring Web MVC, the result is then returned to the client.

Looking at the `toString()` representation from the thread, the following might appear:

```
… INFO 21016 --- [ task-1] ↩
c.t.b.r.SlowRestController  : Thread[task-1,5,main]
```

The thread name is `task-1`. This is a simple thread and no longer a worker thread.

9.12.4 WebAsyncTask

Besides the return `Callable`, the next type is a `WebAsyncTask`.[52] You can consider it an extended `Callable` with more possibilities because a `WebAsyncTask` combines the following elements:

- `Callable`
- Timeout
- Task executor

While `Callable` Spring selects the thread for the return, `WebAsyncTask` gives us the choice of specifying a task executor. In addition, a timeout can be set individually. For construction, the class offers the following constructors:

- `WebAsyncTask(Callable<V> callable)`
- `WebAsyncTask(long timeout, Callable<V> callable)`

---

[52] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/context/request/async/WebAsyncTask.html*

- `WebAsyncTask(Long timeout, AsyncTaskExecutor executor, Callable<V> callable)`
- `WebAsyncTask(Long timeout, String executorName, Callable<V> callable)`

9.12.5 A Handler Method Returns DeferredResult

Another return type is `DeferredResult`.[53] As with `WebAsyncTask`, a timeout can be set, but the thread can be selected even more flexibly. Here's an example:

```
// http://localhost:8080/deferred-async
@RequestMapping( "/deferred-async" )
public DeferredResult<String> deferredResultAsync() {
  var deferredResult = new DeferredResult<String>( 5000 ); // 5 Sek. Timeout
  new Thread( () -> {
    log.info( "{}", Thread.currentThread() );
    try {
      TimeUnit.SECONDS.sleep( 2 );
    }
    catch ( InterruptedException e ) {
      throw new RuntimeException( e );
    }
    deferredResult.setResult( "done" );
  } ).start();
  return deferredResult;
}
```

The example uses a new thread, but, of course, you could use a thread pool.

An object of type `DeferredResult` can be thought of as a container. Once the thread has determined the result, `setResult(…)` will set the result. This is a bit comparable to `Future`, which also gets data at some point. Therefore, the class name fits perfectly: “deferred result.”

Another possibility of `DeferredResult` are callbacks that can be set when the operation completes, when errors occur, or when there is a timeout: `onCompletion(Runnable callback)`, `onError(Consumer<Throwable> callback)`, and `onTimeout(Runnable callback)`.

9.12.6 StreamingResponseBody

If byte streams are written, for example, for large images or videos from a database, this can also be done in a thread apart from the worker thread. The controller can supply a `StreamingResponseBody` for these cases (or wrapped in a `ResponseEntity`). The interface looks like this:

---

[53] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/context/request/async/DeferredResult.html*

```
@FunctionalInterface
public interface StreamingResponseBody {

  /**
   * A callback for writing to the response body.
   * @param outputStream the stream for the response body
   * @throws IOException an exception while writing
   */
  void writeTo(OutputStream outputStream) throws IOException;

}
```

Our implementation is passed an `OutputStream` by Spring, and we can write the bytes there. Here is an example:

```
// http://localhost:8080/streaming-response-body
@RequestMapping( "/streaming-response-body" )
public ResponseEntity<?> streamingResponseBodyAsync() {
 StreamingResponseBody streamingResponseBody = out -> {
   out.write( (byte) '[' );
   for ( int i = 0; i < 100; i++ ) {
     String line = "{\"index\": %d, \"value\": %d},%n";
     byte[] bytes = line.formatted( i,
                ThreadLocalRandom.current().nextInt() ).getBytes();
     out.write( bytes );
     out.flush();
     try { TimeUnit.MILLISECONDS.sleep( 100 ); }
     catch ( InterruptedException e ) { }
   }
   out.write( (byte) ']' );
 };
 return
  ResponseEntity.ok()
   .header(HttpHeaders.CONTENT_DISPOSITION,"attachment;filename=random.json")
   .contentType( MediaType.APPLICATION_OCTET_STREAM )
   .body( streamingResponseBody );
}
```

The example builds a JSON document and manually writes the JSON objects to the output stream with a small delay. The client will get the JSON document as a file that it can download.

9.13 Spring Data Web Support

Almost all controllers need to access the repository for a RESTful web service to fetch and store data from there. Often projects use *Spring Data* and Spring Web together, and then there are some interesting shortcuts. The combination of Spring Data and Spring Web is what we call *Spring Data Web Support*.

9.13.1 Loading from the Repository

Many REST endpoints always look the same: There is `@GetMapping` to an ID, and an entity bean is loaded via this ID in the path variable:

```
@GetMapping( "/{id}" )
public Profile get( @PathVariable long id ) {
  return profiles.findById( id ).get();
}
```

In this case, it can be abbreviated:

```
@GetMapping( "/{id}" )
public Profile get( @PathVariable("id") Profile profile ) {
  return profile;
}
```

In the parameter list of the handler method, the type of an entity bean appears, and again it's the `ConversionService` that fetches the entity bean from the repository using `DomainClassConverter`.[54]

When a handler method's parameter is annotated with `@PathVariable` or `@RequestParam` and has a data type that matches an entity bean, such as `Profile`, Spring follows this procedure:

1. Automatically identifies the corresponding repository (in this case, `ProfileRepository`)
2. Extracts the ID from the path variable or query parameter
3. Retrieves the entity bean using the `findById(…)` method (or its equivalent)
4. Initializes the parameter variable (in this case, `profile`) with the retrieved entity bean

If there is no `Profile` for the ID, the parameter variable isn't set to `null`, but Spring Web MVC responds with a 500 error. If this isn't desired, we can use `Optional<Profile>` and produce a 404 ourselves if no profile could be loaded:

---

[54] *https://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/repository/support/DomainClassConverter.html*

```
public ResponseEntity<?> get(
    @PathVariable("id") Optional<Profile> maybeProfile ) {
  return ResponseEntity.of( maybeProfile.map( profileMapper::convert ) );
}
```

9.13.2 Pageable and Sort as Parameter Types

A handler method can accept parameters of type `Pageable` and/or `Sort`. This allows pagination or sort information to be passed to a handler method from the outside.

Following are the possible parameter lists:

- `profiles( Pageable pageable )`
- `profiles( Sort sort )`
- `profiles( Pageable pageable, Sort sort )`
- `profiles( Sort sort, Pageable pageable )`

A `Pageable` and a `Sort` at the same time are less useful because a `Pageable` can contain a sort criterion.

Besides `Pageable` and `Sort`, there are, of course, still all possibilities to capture query parameters, the header, or the body in a handler method.

Process Pageable and Sort

If a handler method receives a `Pageable` or `Sort` object, these objects could be passed directly to the corresponding repository methods. A `JpaRepository`, specifically the base type `[List]PagingAndSortRepository`, provides `findAll(…)` methods that can accept a `Pageable` or `Sort`. This gives us a short path, as the example shows:

```
@GetMapping
public Stream<ProfileDto> profiles( Pageable pageable ) {
  return profiles.findAll( pageable ).map( profileMapper::convert ).get();
  //              ------ Page ------  ----------- Page -----------
}
```

The return of `findAll(…)` is `Page`, and with `map(…)`, we can map the elements to a `Page<ProfileDto>`, ask for a `Stream`, and return that.

Note: Design Question

The controller directly calls the repository, and, normally, you would place a service layer in between. However, this creates another problem: `Pageable` and `Sort` data types appear within the controller—would you want to pass them on to the methods of a service layer? This would mean that the controller "knows" data types from Spring Data, as does the core of the service layer and, therefore, the repositories as well. This doesn't feel clean.

Query Variables for Parameter Types

For Spring Web MVC to build `Sort` and `Pageable` objects for the handler methods, the `page`, `size`, and `sort` query variables are read:

- **page (Pageable)**
  Page number that starts at 0 with a default of 0.
- **size (Pageable)**
  Number of elements on the page, with a default of 20.
- **sort (Pageable/Sort)**
  String with field names, as well as possible sort directions `asc` and `desc`.

Following are four examples of how to write valid calls:

- `?page=2`
- `?page=2&size=5`
- `?page=2&size=5&sort=manelength,desc`
- `?page=2&size=5&sort=manelength,desc&sort=nickname,asc`

@PageableDefault parameterized Pageable

If nothing is passed, default values take effect. Because these don't have to be optimal, there is a special annotation, `@PageableDefault`,[55] which can be used to declaratively control the page and the number of elements of a page. Here's an example:

```
@GetMapping
public Stream<ProfileDto> profiles(
  @PageableDefault( size = 5, sort = "nickname" ) Pageable pageable
) {
  return profiles.findAll( pageable )
                 .map( profileMapper::convert ).get();
}
```

When we use annotation `@PageableDefault` this way, it means that we've set default values for pagination in our application. Specifically, it indicates that each page will contain five elements, and the sorting criterion will be based on the `nickname`.

The default values are directly in the declaration:

```
@Documented @Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.PARAMETER)
public @interface PageableDefault {
  int value() default 10;
  int size()  default 10;
  int page()  default 0;
```

---

[55] *https://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/web/PageableDefault.html*

```
  String[] sort() default {};
  Direction direction() default Direction.ASC;
}
```

There are some configuration properties with prefix `spring.data.web.pageable`, as listed at *https://docs.spring.io/spring-boot/docs/current/reference/html/application-properties.html#application-properties.data*.

@SortDefault Parameterized Sort

Besides `@PageableDefault`, annotation `@SortDefault`[56] is used to specify a default value for the `Sort` object. This annotation isn't as common because `@PageableDefault` can also contain sort criteria. Here's an example:

```
@GetMapping
public Stream<ProfileDto> profiles(
  @SortDefault.SortDefaults({
    @SortDefault( "birthdate" ),
    @SortDefault( "nickname" ),
  }) Pageable pageable
)
```

The example shows that the special container `@SortDefaults` can be used to specify multiple sorting criteria.

9.13.3 Return Type Page

We've already looked at an example with pagination. In this case, the `findAll(…)` method was passed a `Pageable` implementation, and the results were converted to DTOs and returned as a `Stream`. However, a Spring Web MVC method is allowed to return a `Page` directly:

```
@GetMapping
public Page<ProfileDto> profiles( Pageable pageable ) {
  return profiles.findAll( pageable ).map( profileMapper::convert );
}
```

This simplifies the code even more.

We just have to keep in mind that the return is now different because the JSON document contains metadata about the page, such as the page number and the number of elements.

---

[56] *https://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/web/SortDefault.html*

Consumers invoking these RESTful web services must be prepared for this change in return.

9.13.4 Querydsl Predicate as a Parameter Type

Filter and sorting criteria are necessary in many applications. The task is to translate the criteria encoded in the URL into a database query. Here’s an example:

```
@GetMapping( "/search" )
public Page<ProfileDto> search(
  @RequestParam(name = "nickname", required = false) String nickname,
  @RequestParam(name = "gender", required = false) Integer gender,
  @RequestParam(name = "page", defaultValue = "0") int page,
  @RequestParam(name = "size", defaultValue = "50") int size
) {
  BooleanBuilder booleanBuilder = new BooleanBuilder();

  if ( StringUtils.hasText( nickname ) )
    booleanBuilder.and( QProfile.profile.nickname.eq( nickname ) );
  // …

  Predicate predicate = booleanBuilder;
  Pageable pageable = PageRequest.of( page, size, Sort.by( "id" ) );

  return profiles.findAll( predicate, pageable )
                 .map( profileMapper::convert );
}
```

The first half of the parameter list accepts the criteria, and the last two parameters accept the values for pagination. The highlighted area translates the parameters into `Predicate` and `Pageable` for the query.

This can be simplified. The first shortcut is that `page` and `size` aren’t necessary because this can be captured by a `Pageable` in the parameter list. The second shortcut is that the transfer of the query parameters to a Querydsl `Predicate` can be taken over by Spring Web MVC. It looks like this:

```
@GetMapping( "/search" )
public Page<ProfileDto> search(
    @QuerydslPredicate( root = Profile.class ) Predicate predicate,
    @PageableDefault( size = 50, sort = "id" ) Pageable  pageable ) {
  return profiles.findAll( predicate, pageable )
                 .map( profileMapper::convert );
}
```

The simplification is striking. Spring Web MVC directly provides us with a `Predicate` and `Pageable` object, which can be passed to the `findAll(…)` methods without detours. Possible calls include the following:

- *http://localhost:8080/api/profiles/search?size=5&nickname=FillmoreFat*
- *http://localhost:8080/api/profiles/search?gender=2&hornlength=10&page=0&size=5*
- *http://localhost:8080/api/profiles/search*

One thing remains to be noted: the queries are based on equality. This is fine for `gender` or `attractedToGender`, but less so for `manelength` or `lastseen`, where ranges are more relevant. A solution is provided by `QuerydslBinderCustomizer`[57] from Spring Data. This allows user-defined bindings for Querydsl predicates to be configured. The binding rules determine how request parameters are translated into Querydsl predicates. For example, min-max ranges can be defined for mane length, or string comparisons can be configured independently of case, and so on.

`QuerydslBinderCustomizer` declares a method `void customize(QuerydslBindings bindings, T root)`, where `T extends com.querydsl.core.types.EntityPath<?>`. The interface is generally implemented in the repository interface via a default method. Here's an example:

```
public interface ProfileRepository extends JpaRepository<Profile, Long>,
      QuerydslPredicateExecutor<Profile>, QuerydslBinderCustomizer<QProfile> {
  @Override
  default void customize( QuerydslBindings bindings, QProfile profile ) {
    MultiValueBinding<NumberPath<Short>,Short> bindingManelengthBetween=
(path,value)->{
      List<? extends Number> values = new ArrayList<>( value );
      short minManelength = values.get( 0 ).shortValue();
      if ( values.size() == 1 )
        return Optional.of( path.goe( minManelength ) );
      else {
        short maxManelength = values.get( 1 ).shortValue();
        return Optional.of( path.between( minManelength, maxManelength ) );
      }
    };
    bindings.bind( profile.manelength ).all( bindingManelengthBetween );
  }
}
```

---

[57] *https://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/querydsl/binding/QuerydslBinderCustomizer.html*

In the `customize(…)` method, a new binding for `manelength` is defined. A valid REST call could look like this:

http://localhost:8080/api/profiles/search?manelength=1&manelength=100

While the repository code was previously just an interface without code, now infrastructure code comes into the onion core, which is a disadvantage in this implementation.

9.14 Documentation of a RESTful API with OpenAPI

When an HTTP client wants to connect to an endpoint, various information must be known:

- Under which path an endpoint is located
- Which path variables are necessary
- Which query parameters are evaluated
- Whether a JSON document is sent via a `POST` or `PUT`
- How the exchange objects are constructed
- Which status codes to expect
- Whether operations are asynchronous

Endpoints can't be used without documentation. While one could document the endpoints using various methods such as Word documents, screenshots, or a wiki, the problem with these approaches is that the implementation and documentation can quickly become out of sync. To avoid this issue, a standardized description that ensures implementation and documentation stay aligned is highly recommended.

9.14.1 Description of a RESTful API

Various standards exit for documenting and describing RESTful web services:

- OpenAPI Specification (*https://spec.openapis.org/oas/latest.html*)
- RESTful API Modeling Language (RAML) (*https://raml.org*)
- API Blueprint (*https://apiblueprint.org*)

Out of the three, the OpenAPI specification holds the most significance. It originated from the Swagger specification, which was first introduced in early 2011. The OpenAPI initiative is supported by the Linux Foundation.

9.14.2 OpenAPI Specification

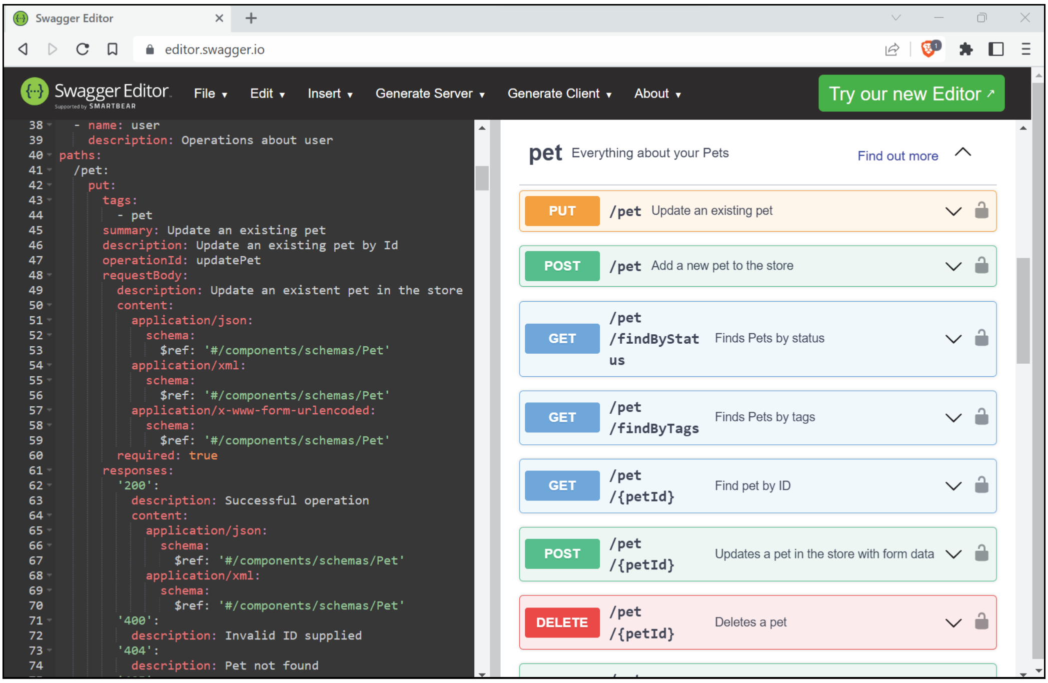
The mission of the OpenAPI specification is well summarized on their website (*https://swagger.io/specification/*):

> *The OpenAPI Specification (OAS) defines a standardized, language-independent interface for RESTful APIs that enables both humans and computers to discover and understand the capabilities of the service without accessing source code, documentation, or network traffic. When properly defined, a consumer can understand and interact with the remote service with a minimum of implementation logic.*

An OpenAPI Document in the Swagger Editor

The OpenAPI specification revolves around defining a document that outlines the API. The document serves as a detailed description of the API and can be written in either JSON or YAML format. Those who have experience with SOAP-based web services may be familiar with Web Services Description Language (WSDL), which is similar in function to the OpenAPI document. The OpenAPI document allows developers to easily understand the functionalities and structure of the API, making it easier to work with and integrate into their applications.

To get a first impression of what an OpenAPI document looks like, you can call the interactive editor at *https://editor.swagger.io/* (see Figure 9.10).



Swagger Editor
editor.swagger.io
Swagger Editor
Supported by SMARTBEAR
File
Edit
Insert
Generate Server
Generate Client
About
Try our new Editor
pet Everything about your Pets
Find out more
PUT /pet Update an existing pet
POST /pet Add a new pet to the store
GET /pet/findByStatus Finds Pets by status
GET /pet/findByTags Finds Pets by tags
GET /pet/{petId} Find pet by ID
POST /pet/{petId} Updates a pet in the store with form data
DELETE /pet/{petId} Deletes a pet

```
  - name: user
    description: Operations about user
paths:
  /pet:
    put:
      tags:
        - pet
      summary: Update an existing pet
      description: Update an existing pet by Id
      operationId: updatePet
      requestBody:
        description: Update an existent pet in the store
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/Pet'
          application/xml:
            schema:
              $ref: '#/components/schemas/Pet'
          application/x-www-form-urlencoded:
            schema:
              $ref: '#/components/schemas/Pet'
        required: true
      responses:
        '200':
          description: Successful operation
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/Pet'
            application/xml:
              schema:
                $ref: '#/components/schemas/Pet'
        '400':
          description: Invalid ID supplied
        '404':
          description: Pet not found
```

Figure 9.10 Swagger Editor with a Sample Document

The document is displayed live with all changes directly visible.

9.14.3 Where Does the OpenAPI Document Come From?

An OpenAPI document plays a crucial role in providing a clear description of how an HTTP client can access a web service. There are two approaches to obtaining this document:

- **Specifications-first (or spec-first)**
  This approach involves creating the OpenAPI document from scratch using specialized editors such as the Swagger Editor or plugins for popular development environments. This approach allows developers to define the API functionalities and structure before writing any code.
- **Code-first (or implementation-first)**
  This approach starts with, for example, Java code, and an OpenAPI document is generated at runtime using a tool. This approach allows developers to focus on writing code first and generating documentation automatically, but it *may* result in a less detailed and less precise documentation compared to the specifications-first approach.

Both variants have their advantages and disadvantages, as listed here:

- **Specification-first**:
  - If the target is an OpenAPI document, direct development is faster than if the document has to be generated first.
  - Mocking is supported by tools. In other words, fake servers can be generated automatically, which can be used to develop clients.
  - The requirements are abstract and not a “waste product” from the code.
  - Merging is difficult when working on complex OpenAPI documents in multiple places at the same time.
  - The process is code-intensive, such as in the schema definition for DTOs.
- **Code-first/implementation-first**:
  - You can quickly get from the implementation to the OpenAPI document.
  - It’s unnecessary to learn the OpenAPI specification and the YAML format. Therefore, this approach is easy for developers.
  - It always takes a build before you can retrieve the OpenAPI document.
  - Changes to the OpenAPI document can only be implemented via code and require a build.

9.14.4 OpenAPI with Spring

We want to run through both solutions, starting with the code-first approach. At the center is a set of open-source libraries from Swagger (*https://swagger.io*), from which, for example, annotation types, code generators, and the graphical *Swagger UI* originate. The code is maintained by SmartBear Software.

Swagger can be integrated into a Spring application to generate OpenAPI documents. There are three projects that accomplish this:

- springdoc-openapi (*https://springdoc.org*), very active
- OpenAPI v3 generator for Spring Boot (*https://github.com/qaware/ openapi-generator-for-spring*), active
- SpringFox (*https://springfox.github.io/springfox*), inactive since 2020

The origin of all these libraries is SpringFox, however, this project is dead today, so we'll use springdoc.

9.14.5 springdoc-openapi

To use springdoc, a dependency is required. Therefore, we enter the following into the POM:

```
<dependency>
  <groupId>org.springdoc</groupId>
  <artifactId>springdoc-openapi-starter-webmvc-ui</artifactId>
  <version>2.2.0</version>
</dependency>
```

In fact, `springdoc-openapi-starter-webmvc-ui`[58] offers a bit more. We'll take a look at why that is in a moment.

Generated OpenAPI Documents

After including the project, we can have OpenAPI documents generated automatically—we don't have to do anything for that. Springdoc picks out all the metadata, pulling in other information such as paths, HTTP methods, parameter types, *Jakarta Bean Validation* annotations, and `@ResponseStatus` codes.

Springdoc provides two new endpoints under which we can retrieve the OpenAPI document as JSON or YAML:

- JSON: *http://localhost:8080/v3/api-docs*
- YAML: *http://localhost:8080/v3/api-docs.yaml*

The path can be changed via `springdoc.api-docs.path=...`.

By default, springdoc examines all controllers and generates the document. To omit endpoints, you can either use `@Hidden`[59] to mark individual controllers or set configuration properties. Here's an example:

---

[58] Formerly `org.springdoc:springdoc-openapi-ui`.

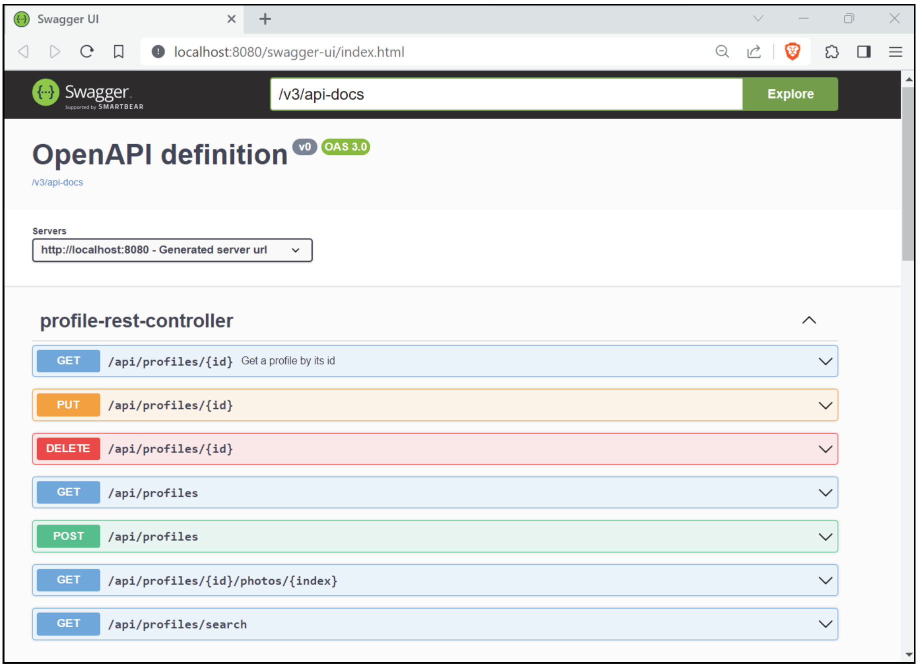
[59] *https://docs.swagger.io/swagger-core/v2.1.1/apidocs/io/swagger/v3/oas/annotations/Hidden.html*

```
springdoc.packagesToScan=com.tutego.date4u.secure, example.com
springdoc.pathsToMatch =/api/stats/**, /api/billing/**
```

This tells which packets are scanned or which paths must match for the documentation to be generated.

Swagger UI

The generated output in JSON or YAML is good for tools, but not optimally readable for humans. To allow API users to explore the endpoints directly, the Swagger UI is available (see Figure 9.11). We had already included it as a dependency. The website is *http://localhost:8080/swagger-ui.html*.



OpenAPI definition v0 OAS 3.0

/v3/api-docs

Servers: http://localhost:8080 - Generated server url

profile-rest-controller

- GET /api/profiles/{id} Get a profile by its id
- PUT /api/profiles/{id}
- DELETE /api/profiles/{id}
- GET /api/profiles
- POST /api/profiles
- GET /api/profiles/{id}/photos/{index}
- GET /api/profiles/search

Figure 9.11 Swagger UI Displaying REST Endpoints Interactively

Calls can actually be made on the web page, and so Postman or IntelliJ's HTTP client can be substituted in simple cases.

There are several things you can enable and control with Swagger UI, such as whether Swagger UI is basically enabled or not and whether real calls may be made or are read-only:

```
springdoc.swagger-ui.enabled=true
springdoc.swagger-ui.path =/swagger-ui.html
springdoc.swagger-ui.tryItOutEnabled=false
springdoc.swagger-ui.filter=false
springdoc.swagger-ui.syntaxHighlight.activated=true
```

9.14.6 Better Documentation with OpenAPI Annotations

The springdoc extension uses all available metadata to construct an OpenAPI document. However, there are certain program elements that can't be read by springdoc, such as a brief description, sample data, header and body information, and status codes from dynamic `ResponseEntity` returns. To address this issue, developers can use special annotations (*https://javadoc.io/doc/io.swagger.core.v3/swagger-annotations/latest/index.html*) to add supplementary information to the endpoints.

Here's an example:

```
@Operation( summary = "Get a profile by its id" )
@ApiResponses( {
    @ApiResponse( responseCode = "200", description = "Profile found",
        content = { @Content( mediaType = "application/json",
            schema = @Schema( implementation = ProfileDto.class ) ) } ),
    @ApiResponse( responseCode = "400", description = "Invalid id",
        content = @Content ),
    @ApiResponse( responseCode = "404", description = "Profile not found",
        content = @Content ) } )
@GetMapping( "/{id}" )
public ResponseEntity<?> get( @PathVariable long id )
```

Annotation `@Operation` sets a description with `summary`. `@ApiResponses` documents the different responses a client might get.

The OpenAPI specification provides annotations for every feature, enabling developers to fully describe endpoints using these annotations. While this may seem beneficial, there are also some drawbacks to this approach. One issue is that any changes to the OpenAPI document can only be made indirectly through modifications to the Java code. This can be problematic, as even minor mistakes may require a complete rebuild, along with any associated processes.

In addition, using annotations to define endpoints may lead to code that is tightly coupled to the OpenAPI document, making it difficult to separate concerns and modify individual components without affecting the overall functionality. As such, it's important to carefully consider the use of annotations and ensure that they are used in a manner that is sustainable and flexible over time.

That is why there is another way, as we'll get to next.

9.14.7 Generate Java Code from an OpenAPI Document

Another approach is to begin with an OpenAPI document and generate Java code based on it. This specification-first approach involves using a generator tool that automatically creates code from the OpenAPI document. One such tool is the *OpenAPI Generator* (*https://openapi-generator.tech*), which is an actively maintained successor to the `swagger-codegen-maven-plugin`.

The OpenAPI Generator offers a wide range of generators for both client-side and server-side code. To use it with Spring, developers need to configure a plugin in the POM file. This involves specifying the location of the YAML file containing the OpenAPI document, as well as the target package where the generated code should be stored. Additionally, developers can specify other options, such as using Jakarta Bean Validation. The configuration might look something like this:

```
<plugin>
<groupId>org.openapitools</groupId>
<artifactId>openapi-generator-maven-plugin</artifactId>
<version>6.6.0</version>
<executions>
  <execution>
    <goals><goal>generate</goal></goals>
    <configuration>
      <inputSpec>${project.basedir}/src/main/resources/openapi.yaml
      </inputSpec>
      <generatorName>spring</generatorName>
      <library>spring-boot</library>
      <modelNameSuffix>Dto</modelNameSuffix>
      <generateApiTests>false</generateApiTests>
      <generateModelTests>false</generateModelTests>
      <configOptions>
        <!-- https://openapi-generator.tech/docs/generators/spring/-->
        <basePackage>com.tutego.date4u.interfaces.rest</basePackage>
        <modelPackage>com.tutego.date4u.interfaces.rest</modelPackage>
        <apiPackage>com.tutego.date4u.interfaces.rest</apiPackage>
        <configPackage>com.tutego.date4u.interfaces.rest</configPackage>
        <useSpringBoot3>true</useSpringBoot3>
        <interfaceOnly>true</interfaceOnly>
        <useBeanValidation>false</useBeanValidation>
        <openApiNullable>false</openApiNullable>
      </configOptions>
    </configuration>
  </execution>
</executions>
</plugin>
```

Then we have to run the appropriate phase so that the code is generated. The generator creates a Java interface, which we then have to implement. But the Java interface carries exactly the annotations that we previously wrote manually. Here's how you can think of it:

```
@Api(value = "profiles", description = "the profiles API")
public interface ProfilesApi {
  …
  @ApiOperation( value = "Get all profiles.",
                 nickname = "findAllProfiles", notes = "",
                 response = FindAllProfilesResponseDto.class,
                 tags = {} )
  @ApiResponses( value = {
      @ApiResponse( code = 200, message = "OK",
                    response = FindAllProfilesResponseDto.class)} )
  @RequestMapping(
      method = RequestMethod.GET,
      value = "/profiles",
      produces = { "application/json" }
  )
  default ResponseEntity<FindAllProfilesResponseDto> findAllProfiles() {
    getRequest().ifPresent( request -> {
      for ( MediaType mediaType : MediaType.parseMediaTypes(
                                   request.getHeader("Accept")) ) {
        if ( mediaType.isCompatibleWith(
                MediaType.valueOf( "application/json" ) ) ) {
          String exampleString = "{ \"items\" : [ { \"nickname\" : \"nickname\
", \"manelength\" : 0, \"id\" : \"id\" }, { \"nickname\" : \"nickname\", \
"manelength\" : 0, \"id\" : \"id\" } ] }";
          ApiUtil.setExampleResponse( request, "application/json",
                                      exampleString );
          break;
        }
      }
    } );
    return new ResponseEntity<>( HttpStatus.NOT_IMPLEMENTED );
  }
}
```

This makes the OpenAPI document the *single source of truth*, that is, the only true source of RESTful documentation, and the code must follow it.

9.14.8 Spring REST Docs

Spring REST Docs (*https://spring.io/projects/spring-restdocs*) is an alternative project for documenting RESTful web services. It involves writing test cases that generate small *AsciiDoc*[60] documents as a by-product during the test run. AsciiDoc is a markup language for documents that allows for the inclusion of HTTP requests and possible responses, similar to what is seen in IntelliJ's HTTP client.

---

[60] *https://asciidoc.org*

These generated AsciiDoc fragments can be combined into a larger document, which can be translated into different languages. The Spring team provides an example in their guide at *https://spring.io/guides/gs/testing-restdocs/*.

It's important to note that Spring REST Docs requires test cases to generate documentation. This can be intentional to encourage documentation, but it can also be stressful if test cases aren't written.

For those using the code-first approach, generating documentation is relatively straightforward. However, the specification-first approach may be more suitable for larger companies. OpenAPI documents can be stored on a server (hub), allowing client generators to create code in various programming languages based on the same document.

9.15 Testing the Web Layer

As we develop RESTful web services, it's crucial to ensure their proper functioning. While we've already written tests for services and repositories, we must now turn our attention to testing the web layer.

9.15.1 Test QuoteRestController

We still want to return to our simple `QuoteRestController` class, which responds to the `api/quotes` endpoint:

```
@RestController
@RequestMapping( "/api/quotes" )
public class QuoteRestController {

  private final static String[] QUOTES = {
      "Date to be known, not to be liked",
      "Dating is all about the chase. It’s fun!",
      "You can’t blame gravity for falling in love"
  };

  @GetMapping
  public String retrieveQuote( int index ) {
    return QUOTES[ index ];
  }
}
```

What could a test look like for this `QuoteRestController`?

When you think of tests, you'll think of the three steps: *given*, *when*, *then*. Translated into Java code, that would mean: "When the `retrieveQuote(…)` method is called on a `QuoteRestController`, a return is expected." That sounds like it could be checked.

An Attempt

A test might look like this:

```
class QuoteRestControllerTest {

  @Test
  void first_quote_exists() {
    // given
    var quoteRestController = new QuoteRestController();

    // when
    String quote = quoteRestController.retrieveQuote( 1 );

    // then
    Assertions.assertThat( quote ).isNotBlank();
  }
}
```

The absence of the `@SpringBootTest` annotation indicates that the test is intended to be a basic component test. The test method sets up the necessary environment, invokes the target method, and asserts the response.

However, this approach falls short when it comes to testing a controller. Instead of testing the controller itself, this implementation tests the underlying service that the controller may use. Consequently, this test case is inadequate for verifying the functionality of a controller.

What We Really Want to Test

Controllers play a crucial role in web services, as they act as a façade for clients to interact with the business logic. As controllers deal with HTTP, testing them requires checking specific aspects:

- Ensuring that specific paths only respond to certain HTTP methods
- Validating that RESTful services send and receive the correct JSON or XML transfer objects with support for content negotiation
- Ensuring that incorrect body, path variables, query parameters, or header assignments result in correct HTTP status codes and error messages
- Verifying if clients have authenticated correctly and have the correct role

When testing REST endpoints, it's important to remember that these tests aren't related to the business logic, which is typically mocked out. A simple component test, without any special annotation, isn't sufficient to test a controller thoroughly.

9.15.2 Annotation @WebMvcTest

During our initial exploration of test cases, we came across two types of annotations: `@SpringBootTest` and slice tests. The former initializes the entire Spring context and boots up the container, while the latter is a more specialized approach that involves scanning only the types required for a specific test scenario. To test controllers, for instance, we need only controllers, security components, and mock objects for services. This is precisely what the `@WebMvcTest`[61] annotation is designed for in Spring Boot. Here's an example of how it might be used:

```
@WebMvcTest // ( QuoteRestController.class )
class QuoteRestControllerTest {

  @Autowired
  private MockMvc mockMvc;
}
```

Annotation `@WebMvcTest` comes to the test classes and can optionally list the controllers. If you don't name the `Class` object explicitly, all controllers will be instantiated. But this is usually not necessary because usually there is one test case per controller.

Via `@WebMvcTest`, we build an interesting object of type `MockMvc`,[62] which we'll use shortly. You can also build a `MockMvc` object yourself via a `MockMvcBuilders` class if you need more control. For us, however, the preconfigured `MockMvc` object fits the bill.

Occasionally, it's necessary to take certain services with the test. Then, you can add the services either with `@Import`, or you can boot the entire container with `@SpringBootTest` and build a `MockMvc @AutoConfigureMockMvc`.

9.15.3 Writing a Test Method with MockMvc

Let's look at the beginning of a test method:

```
@Test
void first_quote_is_a_string() throws Exception {
  mockMvc.perform( get( "/api/quotes?index=1" ) )
         . …
         ;
}
```

The `MockMvc` type provides a `perform(RequestBuilder)` method, and this method is passed the request. In our case, this is a `GET` request, which we send virtually. The

---

[61] *https://docs.spring.io/spring-boot/docs/current/api/org/springframework/boot/test/autoconfigure/web/servlet/WebMvcTest.html*

[62] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/test/web/servlet/MockMvc.html*

`get(…)`method is statically imported from `MockMvcRequestBuilders`.[63] There you can find many other methods such as `post(…)` or `put(…)`. The path is passed to these methods. Most of the methods return a `MockHttpServletRequestBuilder` object[64] where you can set other properties of the request, for example, accept header, character encoding, sent objects, cookies, headers, a principal for security, and so on.

Returning to the `perform(…)` method that returns a `ResultActions` object, let's focus on the calls to the following:

```
ResultActions andExpect(ResultMatcher matcher) throws Exception
```

These are cascaded, which reminds us a bit of AssertJ. However, this is a DSL developed by Spring that has nothing to do with AssertJ. Let's take a look at an example:

```
@Test
void first_quote_is_a_string() throws Exception {
  mockMvc.perform( get( "/api/quotes?index=1" ) )
         .andDo( print() )
         .andExpect( status().isOk() )
         .andExpect( content().string( startsWith( "Dating" ) ) )
         ;
}
```

The first call `.andDo( print() )` isn't really necessary and only logs the traffic between client and server for better traceability. Next, the status 200 (**OK**) is expected. Then the content must start as a string with the substring `"Dating"`. Here, several methods are imported statically. This is common and leads to better readability.

The `andExpect(ResultMatcher)` method receives a `ResultMatcher` as a parameter. `ResultMatcher` is a functional interface, and the utility class `MockMvcResultMatchers`[65] provides prebuilt instances; we've already seen `status()` and `content(…)`. There are many more, such as reading headers or cookies, or checking JSON or XML paths on the results.

Refactoring with Service, @MockBean

Let's make some modifications to the `QuoteRestController` because it's currently directly retrieving data from an array, which isn't very realistic as it should be referencing a service. We start with the `Quotes` class:

```
@Service
class Quotes {
```

---

[63] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/test/web/servlet/request/MockMvcRequestBuilders.html*

[64] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/test/web/servlet/request/MockHttpServletRequestBuilder.html*

[65] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/test/web/servlet/result/MockMvcResultMatchers.html*

```
  private final static String[] QUOTES = { … }
  String quote( int index ) { return QUOTES[ index ]; }
}
```

In the body of the `QuoteRestController` class, we add access to `quotes`:

```
@Autowired Quotes quote;

@GetMapping
public String retrieveQuote( int index ) {
 return quote.quote( index );
}
```

Now follows the type import into `QuoteRestControllerTest`:

```
@WebMvcTest( QuoteRestController.class )
@Import( Quotes.class )
class QuoteRestControllerTest …
```

In the test, we need to use `@Import` to make the wiring from `Quote` to `QuoteRestController` work. This is because `@WebMvcTest` is a slice test, and, by default, no other types are included except controller classes. This was the first step in refactoring.

Normally, you wouldn't use `@Import` in the test because services in the test case are mocked out. Let's use `@WebMvcTest` with `@MockBean`:

```
@WebMvcTest( QuoteRestController.class )
class QuoteRestControllerTest {

  @Autowired
  private MockMvc mockMvc;

  @MockBean
  private Quotes quotes;

  @Test
  void first_quote_is_a_string() throws Exception {

    given( quotes.quote( 1 ) ).willReturn( "Be happy!" );

    mockMvc.perform( get( "/api/quotes?index=1" ) )
        .andDo( print() )
        .andExpect( status().isOk() )
        .andExpect( content().string( "Be happy!" ) )
    ;
  }
}
```

Through mock stubbing, we indicate that a specific quote follows index `1`. Upon initialization of the controller, the mock object is injected. Later, when we access the endpoint, the controller accesses the mock bean and retrieves the content "`Be happy`" from the stubbing of our mock object.

[+]

Tip

Typically, REST endpoints return JSON documents. These can be tested using JSON paths like this:

```
mockMvc.perform( get( "/api/profiles/{id}", 1 )
                      .accept( MediaType.APPLICATION_JSON ) )
        .andExpect( jsonPath( "$[0].id" ).value( "1" ) )
        .andExpect( jsonPath( "$[1].id" ).value( "2" ) )
        ;
```

[»]

Note

We used a mock for the `ProfileRepository` because that is directly accessed by our controller. However, if you use the combination of Spring Web MVC and Spring Data, and implement the handler method simplified as follows, you'll get an error.

```
public ResponseEntity<?> get(
    @PathVariable("id") Optional<Profile> maybeProfile ) { … }
}
```

The reason is that the necessary Spring-managed beans aren't built using the `@WebMvcTest` slice test.

9.15.4 Test REST Endpoints with the Server

The `MockMvc` data type has the advantage that tests are fast, and no real server is initialized. Spring Web MVC even provides a whole set of mock classes for much deeper servlet operations. For example, the `org.springframework.mock.web` package declares `MockHttpServletRequest` for `ServletRequest` or `MockHttpServletResponse` for `ServletResponse`. These are real classes that have nothing to do with Mockito.

`MockMvc` works with these exact mock objects, so they are prepared with the appropriate content. A server isn't started up, however.

WebEnvironment.RANDOM_PORT

Although mocks can be useful, not all test cases can be performed using them. That's why it's important to test the entire stack by starting up the server. To complete this, the application needs to be started along with the web server. However, starting the web application on a specific port can lead to conflicts when running tests in parallel.

To avoid this, the `@SpringBootTest` annotation has a special attribute called `webEnvironment`:

```
@SpringBootTest( webEnvironment = WebEnvironment.RANDOM_PORT )
```

If it's set to `WebEnvironment.RANDOM_PORT`,[66] the framework chooses any free port.

Certain components, which we'll see soon, are automatically configured with this port. If you want to ask for the port, you can use annotation `@LocalServerPort`[67]—like this:

```
@LocalServerPort private int port;
```

If the port is relevant to a test, you can have it injected that way.

9.15.5 WebTestClient

The `@SpringBootTest` test case starts up the server. One of the classes that the Spring team provides for testing is `WebTestClient`. This class is built on `WebClient`, a data type from the Spring WebFlux project. However, to use `WebTestClient`, a dependency must be added as it's not included in the core Spring starter:

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-webflux</artifactId>
  <scope>test</scope>
</dependency>
```

With the dependency, we can have a `WebTestClient` object injected directly into our code and implement the test through it. Let's take a look at what that looks like:

```
@SpringBootTest( webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT )
class QuoteRestControllerTest2 {

  @Autowired private WebTestClient webTestClient;

  @Test
  void first_quote_is_a_string() {
    webTestClient
        .get()
        .uri( "/api/quotes?index=1" )
      //.header(HttpHeaders.ACCEPT,MediaType.APPLICATION_JSON_VALUE)
        .exchange()
```

---

[66] *https://docs.spring.io/spring-boot/docs/current/api/org/springframework/boot/test/context/SpringBootTest.WebEnvironment.html*

[67] *https://docs.spring.io/spring-boot/docs/current/api/org/springframework/boot/test/web/server/LocalServerPort.html*

```
        .expectStatus()
          .is2xxSuccessful()
        .expectBody( String.class )
          .isEqualTo( "Dating is all about the chase. It’s fun!" );
  }
}
```

The Fluent API makes the test readable. An HTTP call actually takes place here as well, and it doesn’t run through a mock. Of course, the API is very reminiscent of `MockMvc` because the tests are similar. It’s a shame that the APIs differ in detail then, but the types were developed five years apart.

With the `WebTestClient`,[68] there is again a corresponding Java method for the HTTP method. The `get(…)` method, like other methods, returns a `RequestHeadersUriSpec`[69] object, which is a nested type in `WebTestClient`. Optionally, we can set the headers, cookies, and so on.

Method `exchange(…)` triggers the actual call. The return type is `ResponseSpec`, which is again a nested type of `WebTestClient`. Then we expect something, which we express with methods of the same type. It’s worth reading the methods in the Javadoc. This is also important for the JSON paths because, in practice, it’s important to be able to check whether the corresponding data is behind the existing JSON paths.

9.16 Best Practices When Using a RESTful API

When creating RESTful web services, there are several key factors to consider. Naturally, this includes selecting appropriate HTTP methods and status codes. However, in addition to these elements, there are other critical considerations to bear in mind. While certain tasks can be automated by the framework, others will require manual programming efforts.

9.16.1 Jakarta Bean Validation

It’s common for clients to transmit JSON documents, and it’s prudent to assume that these documents may contain errors. Therefore, validation is crucial. Fortunately, Spring can effortlessly incorporate Jakarta Bean Validation, minimizing our workload. The only tasks required are to annotate the exchanged containers and the DTOs with Jakarta Bean Validation and include an `@Valid` in the handler methods during the second stage. Let’s examine an example, starting with the container:

---

[68] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/test/web/reactive/server/WebTestClient.html*

[69] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/test/web/reactive/server/WebTestClient.RequestHeadersUriSpec.html*

```
record ProfileDto(
    @Min( 1 ) Long id,
    @NonNull @Length( min = 10, max = 200 ) String nickname,
    @Past LocalDate birthdate,
    @Positive int manelength,
    @Min( 1 ) int gender,
    @Min( 1 ) Integer attractedToGender,
    String description,
    @Past LocalDateTime lastseen
) { }
```

Conveniently, annotations are also correctly recognized and evaluated for records. To enable validation, the `@Valid` annotation is added to the `@RequestBody` parameter. Here is an example using the `update(…)` method that updates a profile:

```
@PutMapping( … )
public ResponseEntity<?> update(
  @PathVariable long id, @Valid @RequestBody ProfileDto dto
)
```

If the object isn't valid, Spring Web MVC will automatically issue an error message. This error message can be transferred to any custom return using a custom `@ControllerAdvice` and `@ExceptionHandler`—we discussed this in Section 9.10.9.

Note

The library for validation isn't included in Spring Boot Starter Web, so we need to include the dependency in our POM:

```
<dependency>
 <groupId>org.springframework.boot</groupId>
 <artifactId>spring-boot-starter-validation</artifactId>
</dependency>
```

9.16.2 Resource ID

It's crucial for resources to have a unique identifier, typically an ID. However, it's also essential to consider whether the ID used internally by the data set in the database (which is also the key) should be made available externally. When the key is automatically generated continuously, it may be enumerated and potentially exploited from outside. In our case, the profiles begin with an ID of 1 and then increment by 1. This approach is generally efficient for the database and suitable for the application. However, it's not advisable to expose these IDs externally. Similar issues may arise with other natural keys, such as ISBNs or email addresses.

While this may not pose a significant security threat with appropriate permissions, it's still advisable to incorporate some abstraction. For instance, an entity could include a UUID that is used for external queries. This approach ensures that if the database ID changes due to mergers of databases, it doesn't affect the client. Additionally, UUIDs are well distributed, reducing the possibility of them being enumerated. If a database ID gets leaked, it can provide insights into the number of entities or the popularity of a service. For instance, if a unicorn is registered with ID 100 today and another unicorn with ID 101 is registered the next day, it's evident that "I am the only one using this service."

9.16.3 Map Data Transfer Objects

When clients and servers exchange data, it's recommended to use transfer objects or DTOs. These specialized containers are solely designed for data exchange and aren't visible at the business logic or repository level. However, manually programming the mapping of entity objects to DTOs can be a laborious task. Fortunately, there are mapping frameworks available that can perform this task for us. One such framework is *MapStruct* (*https://mapstruct.org*), which as a code generator allows for an automated and efficient transformation between objects. The focus is on *mappers* with mapping rules that are described using annotations. This way, a controller can easily convert between domain objects and DTOs through the conversion service.

To use MapStruct in a Spring application, a dependency is needed in the POM file:

```
<dependency>
  <groupId>org.mapstruct</groupId>
  <artifactId>mapstruct</artifactId>
  <version>1.5.5.Final</version>
</dependency>
```

MapStruct realizes the mapping of objects not through reflection, but through a code generator. For this, an annotation processor must be included in the POM file. Under `<build><plugins>`, include the following:

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-compiler-plugin</artifactId>
  <configuration>
<annotationProcessorPaths>
   <path>
      <groupId>org.mapstruct</groupId>
      <artifactId>mapstruct-processor</artifactId>
      <version>1.5.5.Final</version>
   </path>
</annotationProcessorPaths>
  </configuration>
</plugin>
```

The `mapstruct-processor` analyzes the mapper annotations in the code to generate the necessary implementation classes. We declare the mapper as an interface that extends `org.springframework.core.convert.converter.Converter`:

```
import org.mapstruct.*;
import org.springframework.core.convert.converter.Converter;

@Mapper( componentModel = MappingConstants.ComponentModel.SPRING )
public interface ProfileMapper extends Converter<Profile, ProfileDto> {
  // @Mapping(target = "ABC", source = "RST")
  // @Mapping(target = "DEF", source = "XYZ")
  ProfileDto convert( Profile profile );
}
```

The first annotation `@Mapper` declares the mapping strategy for the frameworks. With several mappers, `@MapperConfig` can refer to a configuration class.[70] We can specify the specific mapping rules via `@Mapping`, but this isn't necessary in our case because the DTO type uses the same identifiers. During the build, the mapper implementation is generated under *target/generated-sources/annotations* in the same package where our mapper was declared. It looks like this:

```
@Generated( … )
@Component
public class ProfileMapperImpl implements ProfileMapper {
  @Override
  public ProfileDto convert(Profile profile) {
    if ( profile == null ) return null;

    Long id = null;
    …
    LocalDateTime lastseen = null;

    id = profile.getId();
    …

    ProfileDto profileDto = new ProfileDto(
     id, nickname, birthdate, manelength, gender,
     attractedToGender, description, lastseen );

    return profileDto;
  }
}
```

---

[70] *https://mapstruct.org/documentation/stable/reference/html/#shared-configurations*

Each state is read from the source and initializes the new object with it. The implementation carries the annotation `@Component` so that Spring Boot automatically recognizes it and registers it in the `ConversionService`.

There are two ways to use the mapper. The first is to inject the mapper directly:

```
@Autowired ProfileMapper profileMapper;
```

Then, the `convert(…)` method is called as before.

The second way is through the conversion service. Because Spring Boot automatically registers each `Converter` within the `ConversionService`, you can have it injected:

```
@Autowired ConversionService conversionService;
```

The `convert(…)` method of the `ConversionService` receives the source object and the target type:

```
var dto = conversionService.convert( profile, ProfileDto.class );
```

With MapStruct, you consequently don't have to write the mappers yourself. MapStruct is very powerful, and it makes sense to carefully study the reference documentation.[71]

9.16.4 Versioning of RESTful Web Services

Web service versioning is a critical consideration as paths, status codes, and the format of exchanged JSON documents can evolve. Numerous approaches exist to tackle these changes. While making everything fundamentally backward compatible may seem like a plausible solution, it's difficult to achieve in practice. Adding fields is generally feasible, but invalidating fields or changing data types may be necessary over time.

The general strategy is to version web services, that is, to assign identifiers for the versions. There are different variants for transmitting the version identifier:

- With a URL path
- By URL parameter
- Via a header

The variant with the path is the most common and can look like this:

- *https://server/api/v1/resource*
- *https://server/api/v1.0/resource*
- *https://server/api/v2022-06/resource*

---

[71] *https://mapstruct.org/documentation/stable/reference/html/#defining-mapper*

The second case shows *semantic versioning* (SemVer) that uses the `Major.Minor.Patch` format. In the third example, we see a year-month format, called *calender versioning* (CalVer). The second variant is useful when the web service has reached a certain stability so that it either doesn't change or is backward compatible.

In code, it may look like this:

```
class ResourceV1 { … }
class ResourceV2 { … }

@RestController
public class VersionRestController {

  @GetMapping( "/v1/resource" )
  public ResourceV1 resourceV1PathVariable() {
    return new ResourceV1( … );
  }

  @GetMapping( "/v2/resource" )
  public ResourceV2 resourceV2PathVariable() {
    return new ResourceV2( … );
  }
}
```

The two versioned endpoints provide `ResourceV1` and `ResourceV2` due to different version numbers. A type relationship between the two resources isn't recommended. That is, `ResourceV(N+1)` shouldn't be a subclass of `ResourceV(N)` because we've already said that data types can change or fields can become invalid.

URI templates with regexes are useful in this context. This allows us to say that there should be a single digit after `v`:

```
@RequestMapping( "/{apiVersion:v\\d}" )
```

9.17 Secure Web Applications with Spring Security

In this section, we'll look at how to secure web applications with *Spring Security*. Spring Security is an important part of the *Spring Framework*, and we'll look at that first.

9.17.1 The Importance of Spring Security

Spring Security plays a critical role in securing applications by performing two essential tasks: *authentication* and *authorization*. Authentication is the process of identifying a user attempting to access the application. This process usually involves asking the user for a username and password, but other means such as fingerprint or digital certificates can also be used for identification.

Once a user is authenticated, the next step is authorization. This involves determining whether the authenticated user is authorized to perform the intended action. Authorization is often referred to as *access control* and is implemented by applying security constraints or rules to resources. The security constraints define which users or roles are allowed to access the resource and what actions they can perform on it.

The reference documentation at *https://docs.spring.io/spring-security/reference/* is very extensive, and we'll learn the important concepts in the context of Spring Web.

9.17.2 Dependency on Spring Boot Starter Security

Spring Security is quickly integrated, and only one dependency is required:

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-security</artifactId>
</dependency>
```

If you add this dependency to a regular Spring Boot starter project without any additional dependencies, you won't notice the existence of Spring Security. However, if you use Spring Boot Starter Security along with Spring Boot Starter Web, a lot happens—many Spring-managed beans are created.

Before we dive deeper into Spring Security in the context of Spring Web, let's discuss basic data types that occur and are relevant in Spring Security.

9.17.3 Authentication

Let's start with how a logged-in user is represented in the system. For this purpose, Spring Security declares the `Authentication` data type. Java SE already has a `Principal` data type for users, and in Spring Security, `Authentication` is a subtype of that data type (see Figure 9.12).

The `Authentication` subtype in Spring Security plays a critical role in the authentication process. It contains a set of *granted authorities*, which represent the roles or rights that a user is granted within the system. These granted authorities are assigned to a user during the authentication process based on the user's credentials and other information, such as user roles or groups.

In addition to the granted authorities, the `Authentication` object also contains the user's username and password.

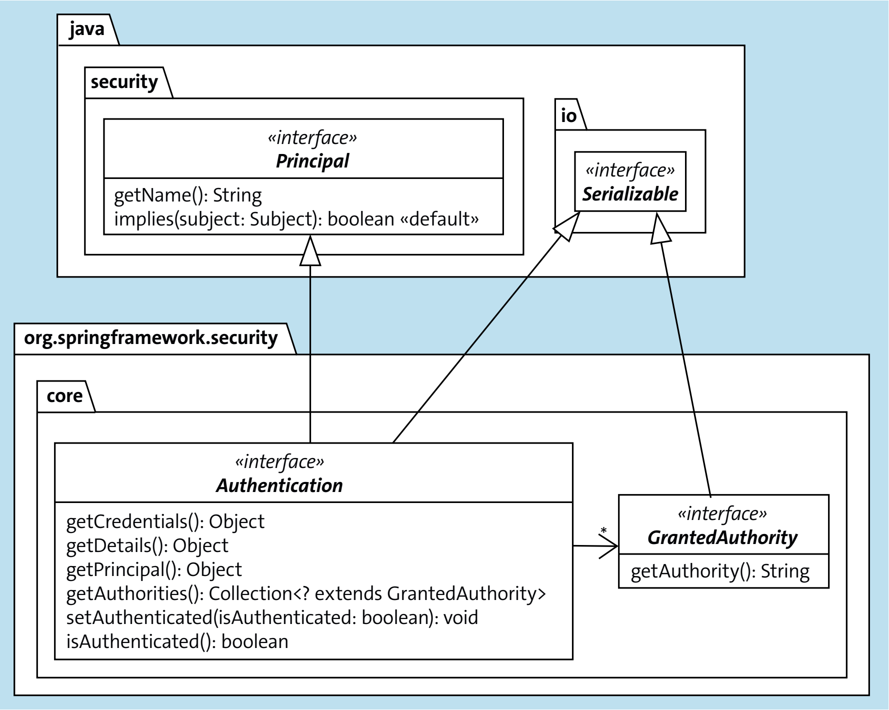

java
security
«interface»
Principal
getName(): String
implies(subject: Subject): boolean «default»
io
«interface»
Serializable
org.springframework.security
core
«interface»
Authentication
getCredentials(): Object
getDetails(): Object
getPrincipal(): Object
getAuthorities(): Collection<? extends GrantedAuthority>
setAuthenticated(isAuthenticated: boolean): void
isAuthenticated(): boolean
*
«interface»
GrantedAuthority
getAuthority(): String

Figure 9.12 "Authentication" Representing Logged-In Users

9.17.4 SecurityContext and SecurityContextHolder

The Spring Security framework stores logged-in users in the system. The `SecurityContext`[72] is responsible for retrieving and storing authentication information, which includes details of the currently authenticated user. The `SecurityContextHolder`[73] class is a helper class that facilitates access to the `SecurityContext`. By default, a thread local object is used to store the security context. This means that the security context is available to methods in the same execution thread.

In Figure 9.13, an accompanying UML diagram shows the data types involved in this process.

The `SecurityContextHolder` class has only static methods. The most important one is `getContext()`, which returns the current `SecurityContext` object. A `SecurityContext` can be used to retrieve and store an `Authentication` object. This way, we can manually authenticate users and store them in the system.

---

[72] *https://docs.spring.io/spring-security/site/docs/current/api/org/springframework/security/core/context/SecurityContext.html*

[73] *https://docs.spring.io/spring-security/site/docs/current/api/org/springframework/security/core/context/SecurityContextHolder.html*

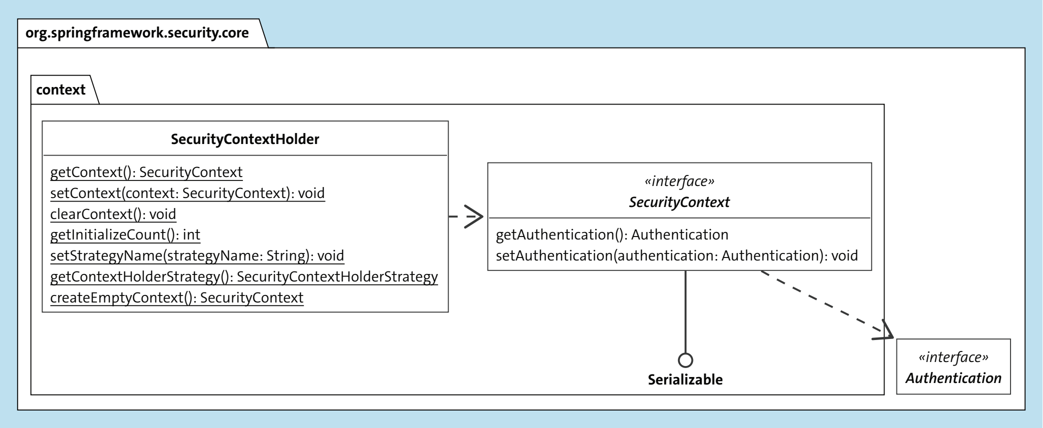

org.springframework.security.core
context
SecurityContextHolder
getContext(): SecurityContext
setContext(context: SecurityContext): void
clearContext(): void
getInitializeCount(): int
setStrategyName(strategyName: String): void
getContextHolderStrategy(): SecurityContextHolderStrategy
createEmptyContext(): SecurityContext
«interface»
SecurityContext
getAuthentication(): Authentication
setAuthentication(authentication: Authentication): void
Serializable
«interface»
Authentication

Figure 9.13 "SecurityContext" and "SecurityContextHolder"

Inquire Logged-In User

If we want to retrieve usernames or other user details, we come to the `SecurityContext` from the `SecurityContextHolder`:

```
Authentication auth = SecurityContextHolder.getContext().getAuthentication();
System.out.println( auth ); // null
```

By default, there are no logged-in users, so the result is `null`.

We've just seen that with the `SecurityContext`, there is also a setter that can be passed in an `Authentication` implementation.

UsernamePasswordAuthenticationToken

`Authentication` is an interface, and an implementing class is `UsernamePasswordAuthenticationToken`[74] (see Figure 9.14).

For testing, we can deposit a user:[75]

```
String principal   = "fillmore.fat@wyman.co";
String credentials = "u87szdzwr6j";
Authentication user =
    new UsernamePasswordAuthenticationToken( principal,
                                             credentials );
SecurityContextHolder.getContext().setAuthentication( user );
```

---

[74] *https://docs.spring.io/spring-security/site/docs/current/api/org/springframework/security/authentication/UsernamePasswordAuthenticationToken.html*

[75] The password is as noted in the database: *https://gist.github.com/ullenboom/2ddaada23508a8ff6d1109d3b8c020d9*.

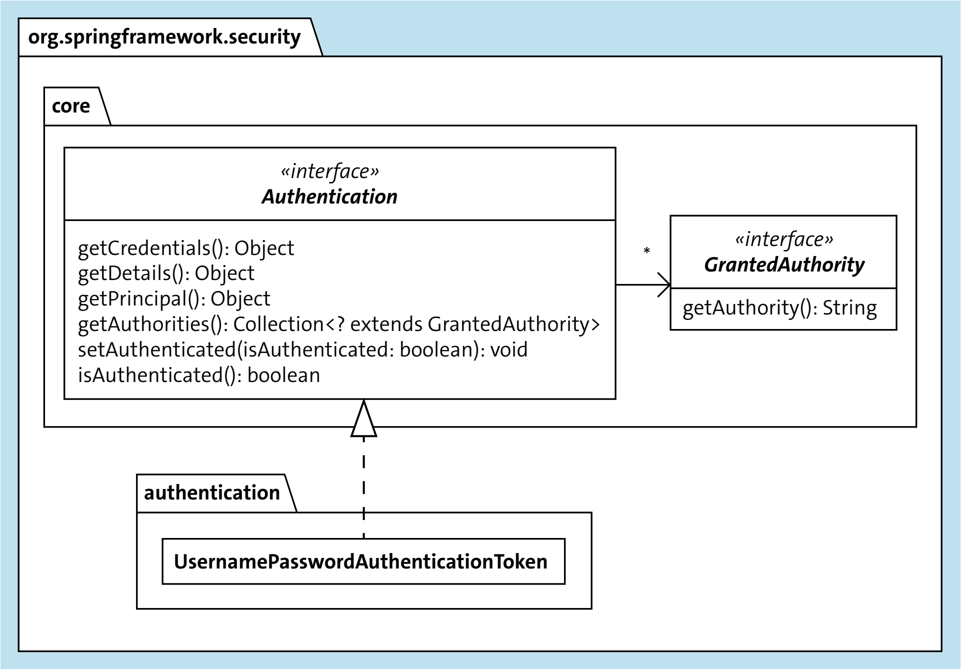

org.springframework.security
core
«interface»
Authentication
getCredentials(): Object
getDetails(): Object
getPrincipal(): Object
getAuthorities(): Collection<? extends GrantedAuthority>
setAuthenticated(isAuthenticated: boolean): void
isAuthenticated(): boolean
*
«interface»
GrantedAuthority
getAuthority(): String
authentication
UsernamePasswordAuthenticationToken

Figure 9.14 "UsernamePasswordAuthenticationToken": An "Authentication" Implementation

The `UsernamePasswordAuthenticationToken` object is initialized with username and password and then set with `setAuthentication(…)`.

Elsewhere in the code—but in the same thread—the `Authentication` object can be requested:

```
System.out.println(
    SecurityContextHolder.getContext().getAuthentication()
);
// UsernamePasswordAuthenticationToken [Principal=fillmore.fat@wyman.co,
// Credentials=[PROTECTED], Authenticated=false, Details=null,
// Granted Authorities=[]]
```

Now the answer is no longer `null` (but it's still `Authenticated=false`).

Manually setting an `Authentication` in the `ContextHolder` is rare, but querying for it's not. However, this is intended to illustrate how the mechanisms work. Later, in Spring Web, a web filter will automatically recognize users and provide an `Authentication` object, which can then be queried in the handler method.

9.17.5 AuthenticationManager

We've seen how to put a user into the `SecurityContext`, but before that, we have to test if the user exists with his name and if the password is correct. This task is done in Spring Security by the type `AuthenticationManager`.[76] This is a functional interface with method `authenticate(…)`, as shown in Figure 9.15.

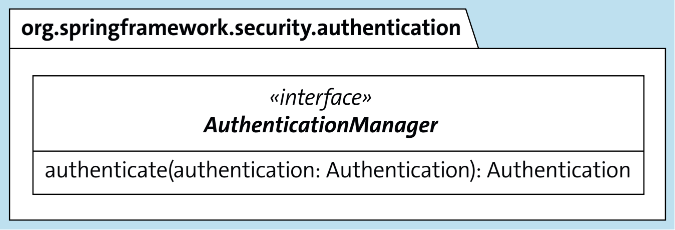

org.springframework.security.authentication
«interface»
AuthenticationManager
authenticate(authentication: Authentication): Authentication

Figure 9.15 "AuthenticationManager" Interface

The `authenticate(…)` method is passed an `Authentication` object, such as a `UsernamePasswordAuthenticationToken` with username and password; an `Authentication` object is returned—at least in the best case. There are two ways the method can respond:

- In the best case, the information in the `Authentication` object is correct; that is, the username is known and the password is correct. Then,, a special flag is set in the returned `Authentication` object, which signals that this user has been authenticated.
- As a last option, it's possible that the `authentication(…)` throws an `AuthenticationException`; that is, the `AuthenticationManager` returns a clear no.

Task: Create a Spring Security Project

We don't bring Spring Security into Date4u so that the security example is isolated and easier to play with. If you want to try this out, create a new Spring Boot project with two dependencies: on Spring Boot Starter Web and Spring Boot Starter Security. Then, the following `DemoRestController` can be integrated into the project:

```
@RestController
public class DemoRestController {
  @GetMapping( "/tip" )
  String shortQuote() { return "Die with memories, not dreams."; }
  @GetMapping( "/stats" )
  String numberOfRegisteredUnicorns() { return "20"; }
}
```

After launching the application, you'll notice that you can no longer access *http://localhost:8080/stats* and *http://localhost:8080/tip*.

---

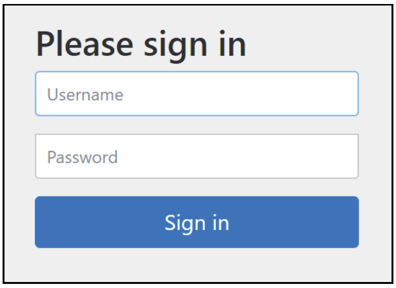
[76] *https://docs.spring.io/spring-security/site/docs/current/api/org/springframework/security/authentication/AuthenticationManager.html*

For example, if you specify `GET localhost:8080/stats` in the IntelliJ HTTP client, the result will look like this:

```
HTTP/1.1 401
Set-Cookie: JSESSIONID=6B18F929C6F6F5A400FE556431992642; Path=/; HttpOnly
WWW-Authenticate: Basic realm="Realm"
…
Content-Length: 0
Date: Mon, 24 Oct 2022 13:44:07 GMT
Keep-Alive: timeout=60
Connection: keep-alive
```

The `401` error stands for `Unauthorized`, which means you can't access both endpoints because the rights are missing.

In the browser, interestingly, the result looks different: there is a dialog like that shown in Figure 9.16 where we can enter a username and password.



Please sign in

Figure 9.16 Attempting to Access a Secured Endpoint Leads to the Login Dialog

Of course, this interactive interface isn't meant for REST endpoints, but if you have dynamically generated web pages, such a dialog is useful.

This behavior is interesting because we had written RESTful web services before, and this dialog never appeared. Therefore, its appearance must be related to Spring Web and Spring Security being deployed together and this query being enabled.

9.17.6 SpringBootWebSecurityConfiguration

As for so many things, in the case of web security and Spring Web, there is an autoconfiguration that is bound to a condition: if we don't make our own security configuration, then the *default web security* applies, and this builds a `SecurityFilterChain` that blocks everything. In code,[77] it looks like this:

---

[77] *https://github.com/spring-projects/spring-boot/blob/main/spring-boot-project/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/security/servlet/SpringBootWebSecurityConfiguration.java*

```
@Configuration( proxyBeanMethods = false )
@ConditionalOnWebApplication( type = Type.SERVLET )
class SpringBootWebSecurityConfiguration {
 /** The default configuration for web security. It relies on Spring
  * Security's content-negotiation strategy to determine what sort of
  * authentication to use. If the user specifies their own
  * {@code WebSecurityConfigurerAdapter} or {@link SecurityFilterChain} bean,
  * this will back-off completely and the users should specify all the bits
  * that they want to configure as part of the custom security configuration.
  */
  @Configuration( proxyBeanMethods = false )
  @ConditionalOnDefaultWebSecurity
  static class SecurityFilterChainConfiguration {
    @Bean @Order( SecurityProperties.BASIC_AUTH_ORDER )
    SecurityFilterChain defaultSecurityFilterChain(HttpSecurity http)
        throws Exception {
      http.authorizeHttpRequests(
        (requests) -> requests.anyRequest().authenticated()
      );
      http.formLogin(withDefaults());
      http.httpBasic(withDefaults());
      return http.build();
    }
  }
  …
}
```

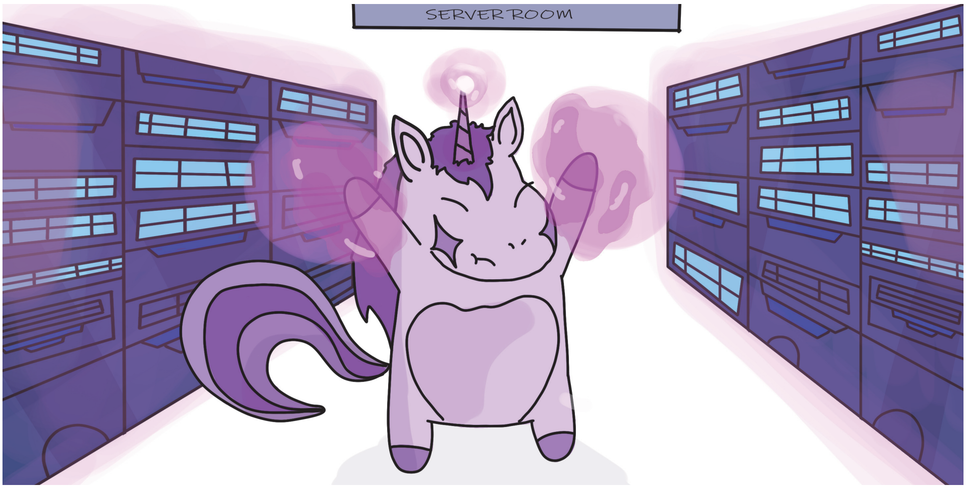
Spring Boot builds a Spring bean of type `SecurityFilterChain`. This is configured so that all incoming requests must be authenticated. There is no exception to this rule. The login is formulated using a form login, and HTTP Basic Authentication is used. This configuration of the `SecurityFilterChain` leads us to the dialog shown earlier in Figure 9.16.

If we want to change this, we just need to build a Spring-managed bean of type `SecurityFilterChain`. Then, `@ConditionalOnDefaultWebSecurity`[78] will no longer hit because, with a custom `SecurityFilterChain`, default web security no longer applies.[79] Behind annotation `@ConditionalOnDefaultWebSecurity` is the `DefaultWebSecurityCondition` that combines `@ConditionalOnClass({ SecurityFilterChain.class, HttpSecurity.class })`, and `@ConditionalOnMissingBean({ SecurityFilterChain.class })`.

---

[78] *https://docs.spring.io/spring-boot/docs/current/api/org/springframework/boot/autoconfigure/security/ConditionalOnDefaultWebSecurity.html*

[79] The `WebSecurityConfigurerAdapter` data type used to exist, and you'll still find a lot about it on the internet. Although the data type has since been removed, the configuration via `SecurityFilterChain` remained unchanged.



A Custom SecurityFilterChain Configuration

The implementation of such a custom configuration class could look like this, for example:

```
@Configuration( proxyBeanMethods = false )
public class SecurityConfig {

  @Bean
  SecurityFilterChain filterChain( HttpSecurity http ) throws Exception {
   http
   .authorizeHttpRequests(
      authorizeHttpRequests ->
        authorizeHttpRequests.requestMatchers( "/tip" ).permitAll()
                                          .anyRequest().authenticated()
    )
    .formLogin( Customizer.withDefaults() )
    .httpBasic( Customizer.withDefaults() );
    return http.build();
  }
}
```

We write ourselves a `@Bean` method that returns a `SecurityFilterChain`[80] object. As an example, we configure it so that access to the */tip* endpoint is permitted, but all other requests must be authenticated. Then, we add the form login and HTTP Basic Authentication, as it was before.

---

[80] *https://docs.spring.io/spring-security/site/docs/current/api/org/springframework/security/config/annotation/web/builders/HttpSecurity.html*

So far, we have two REST endpoints, and we can access */tip*, but all the others, such as */stats*, can only be accessed after login. But what are the credentials?

9.17.7 AuthenticationManager and ProviderManager

Let's return to the `AuthenticationManager`. Its only method, `authenticate(…)`, accepts an `Authentication` object and, in the best case, returns an `Authentication` object with the information that the authentication has succeeded.

This interface is currently implemented by a single class, `ProviderManager`[81] (see Figure 9.17). The `ProviderManager` is a kind of data structure that internally references multiple `AuthenticationProvider`.[82]

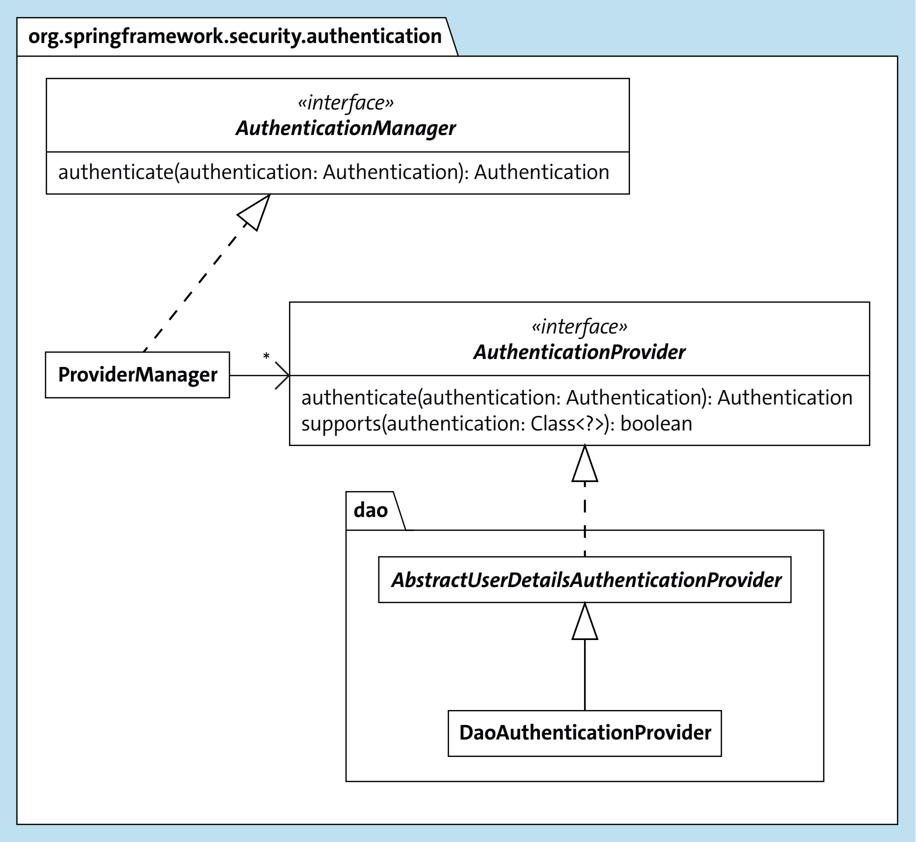

org.springframework.security.authentication
«interface»
AuthenticationManager
authenticate(authentication: Authentication): Authentication
ProviderManager
*
«interface»
AuthenticationProvider
authenticate(authentication: Authentication): Authentication
supports(authentication: Class<?>): boolean
dao
AbstractUserDetailsAuthenticationProvider
DaoAuthenticationProvider

Figure 9.17 "ProviderManager" Referencing "AuthenticationProvider"

---

[81] *https://docs.spring.io/spring-security/site/docs/current/api/org/springframework/security/authentication/ProviderManager.html*

[82] *https://docs.spring.io/spring-security/site/docs/current/api/org/springframework/security/authentication/AuthenticationProvider.html*

An `AuthenticationProvider` is like an `AuthenticationManager`—only with the small difference that the `AuthenticationProvider` has a `supports(…)` method and can thus state whether it supports a certain data type.

A `ProviderManager` contains various `AuthenticationProvider` instances, and these are tested in sequence for a request, as is known from the *chain of responsibility* design pattern. If one of the `AuthenticationProviders` agrees, it takes over the authentication.

The `AuthenticationProvider` interface has around 20 different implementations, each with its own specific functionality. One such implementation is the `DaoAuthenticationProvider`.[83] This provider internally uses the following:

- **`UserDetailsService`**
  This is responsible for retrieving user information from the application's database or other (usually) persistent storage.
- **`PasswordEncoder`**
  This helps to match the provided username and password with the stored ones.

`DaoAuthenticationProvider` checks whether the provided username exists via `UserDetailsService` and retrieves the user's details, such as the encoded password and any associated roles or permissions. It then uses `PasswordEncoder` to verify that the provided password matches the stored password. If authentication is successful, it returns a fully populated `Authentication` object with the user's details and granted authorities.

Tip

If this is of interest to you, you'll benefit from studying the source code of the class.[84]

9.17.8 UserDetailsService Interface

Because `DaoAuthenticationProvider` is automatically registered, and it automatically interacts with `UserDetailsService`, we should take a look at what the `UserDetailsService`[85] has for a task. The UML diagram in Figure 9.18 shows the interfaces `UserDetailsService` and `UserDetails`.

Functional interface `UserDetailsService` has a method called `loadUserByUsername(String)`. When we pass it a username, the result is a `UserDetails` object that represents a user with information such as his password and his roles or rights. What exactly is a username in this case is a matter of definition: it could really be a username, but it could also be an email address or a UUID that identifies a user in the system.

---

[83] *https://docs.spring.io/spring-security/site/docs/current/api/org/springframework/security/authentication/dao/DaoAuthenticationProvider.html*

[84] *https://github.com/spring-projects/spring-security/blob/main/core/src/main/java/org/springframework/security/authentication/dao/DaoAuthenticationProvider.java*

[85] *https://docs.spring.io/spring-security/site/docs/current/api/org/springframework/security/core/userdetails/UserDetailsService.html*

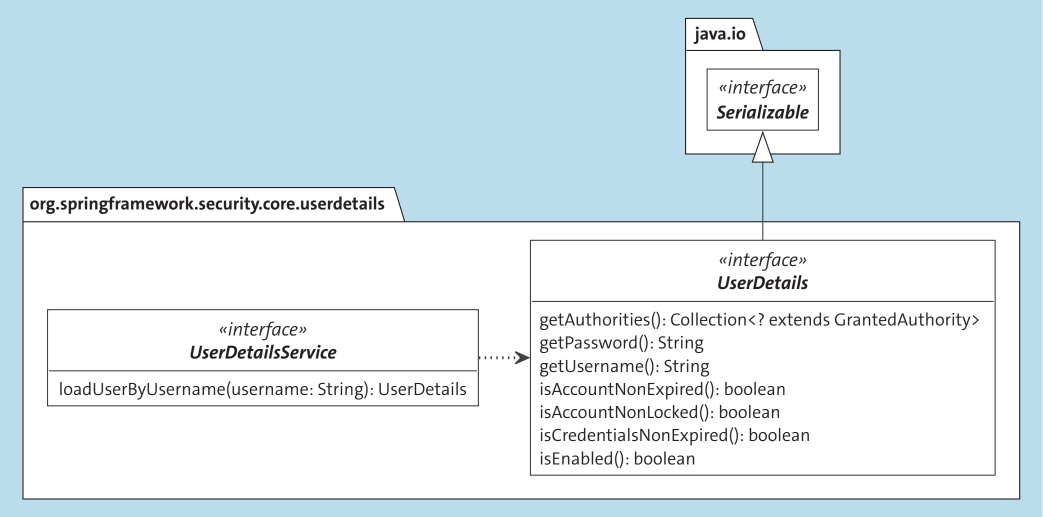

java.io
«interface»
Serializable
org.springframework.security.core.userdetails
«interface»
UserDetailsService
loadUserByUsername(username: String): UserDetails
«interface»
UserDetails
getAuthorities(): Collection<? extends GrantedAuthority>
getPassword(): String
getUsername(): String
isAccountNonExpired(): boolean
isAccountNonLocked(): boolean
isCredentialsNonExpired(): boolean
isEnabled(): boolean

Figure 9.18 "UserDetailsService" Functional Interface

If no user can be found, there is a `UsernameNotFoundException`,[86] which is a subclass of `AuthenticationException`.

> **Note**
>
> The reason for having both `UserDetails` and `Authentication` data types may be unclear because they contain similar information. However, `UserDetails` only serves as a data type for a user who has been fetched from a data source, and it's no longer necessary once the username and password have been verified. In contrast, an `Authentication` object represents the user who has successfully logged in, and it's a long-living object.

UserDetailsServiceAutoConfiguration

Spring Security builds an implementation of `UserDetailsService` through the autoconfiguration of `UserDetailsServiceAutoConfiguration`[87] (specifically, `InMemoryUserDetailsManager`) and registers it. By default, this implementation recognizes a user named `"user"`. The password is generated as a new UUID every time the program starts. This is what it looks like on the console:

```
Using default security password: c8be15de-4488-4490-9dc6-fab3f91435c6
```

---

[86] *https://docs.spring.io/spring-security/site/docs/current/api/org/springframework/security/core/userdetails/UsernameNotFoundException.html*

[87] *https://docs.spring.io/spring-boot/docs/current/api/org/springframework/boot/autoconfigure/security/servlet/UserDetailsServiceAutoConfiguration.html*

Those who use dev tools with the development server will also always see a new password after each restart.

In principle, username and password can also be set, like this:

```
spring.security.user.name=fillmore.fat@wyman.co
spring.security.user.password={noop}u87szdzwr6j
```

Of course, this is only useful for development and testing.

With these login details, we can actually log in to our website. After logging in, the server and browser exchange a cookie, and there is no need to log in again. At *http://localhost:8080/logout*, you can log out again.

UserDetailsService and Selected Implementations

The auto-configuration builds a `UserDetailsService`, namely the `InMemoryUserDetailManager`, which can represent any number of users in memory. The UML diagram from Figure 9.19 illustrates the implementations.

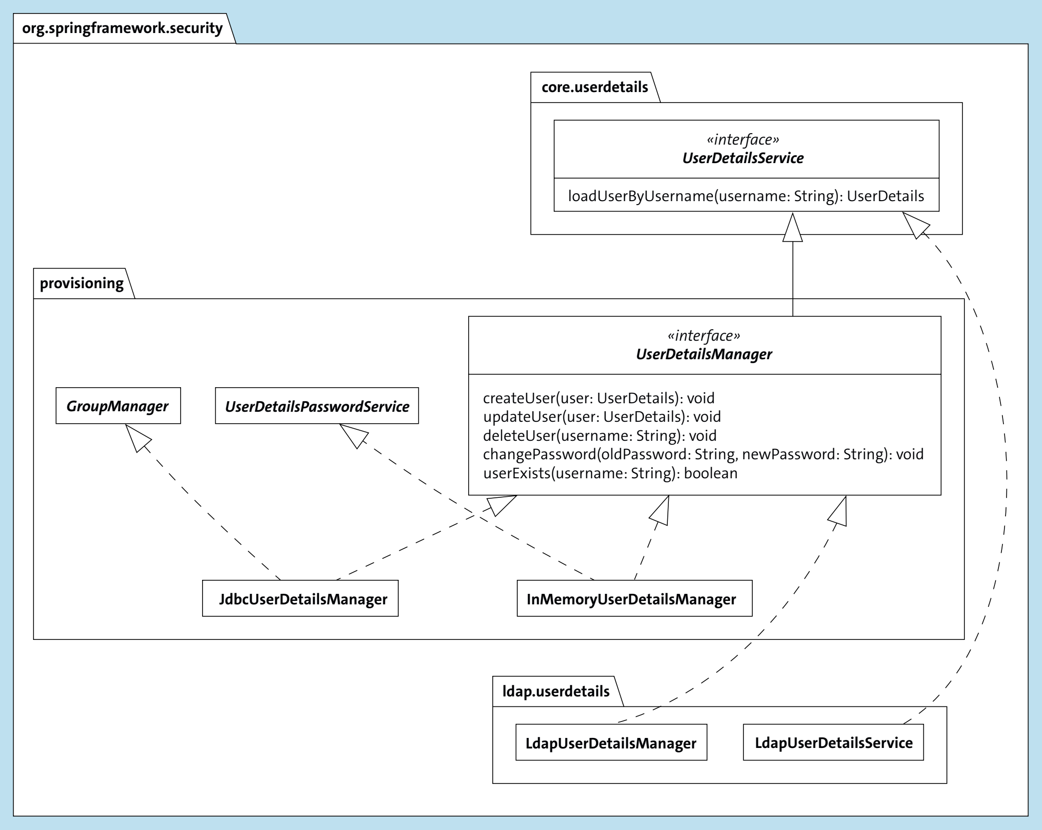

org.springframework.security
core.userdetails
«interface»
UserDetailsService
loadUserByUsername(username: String): UserDetails
provisioning
«interface»
UserDetailsManager
createUser(user: UserDetails): void
updateUser(user: UserDetails): void
deleteUser(username: String): void
changePassword(oldPassword: String, newPassword: String): void
userExists(username: String): boolean
GroupManager
UserDetailsPasswordService
JdbcUserDetailsManager
InMemoryUserDetailsManager
ldap.userdetails
LdapUserDetailsManager
LdapUserDetailsService

Figure 9.19 Subtypes of "UserDetailsService"

Interface `UserDetailsService` has a subtype `UserDetailsManager` that can also be used to create, update, and delete users. You can also use it to change the password and ask if the user exists. There are other implementations such as `JdbcUserDetailsManager`, which works on predefined database tables. If you work with LDAP, you can use a `LdapUserDetailsManager`. The methods that the `UserDetailsManager` adds to the `UserDetailsService` are never used by the Spring Framework itself.

Write Your Own UserDetailsService

Writing your own `UserDetailsService` is easy and often happens in practice. We only need to implement the interface, that is, realize method `loadUserByUsername(…)`:

```
@Configuration( proxyBeanMethods = false )
public class UserDetailsServiceConfig {

 @Bean
 UserDetailsService userDetailsService() {
  return new UserDetailsService() {
   @Override public UserDetails loadUserByUsername( String username ) {
      return switch ( username ) {
        case "fillmore.fat@wyman.co" ->
           User.withUsername( username )
               .password( "{noop}u87szdzwr6j" )
               .roles()
               .build();
        case "candy.kane@mills.info" ->
           new User( username, "{noop}mk8suwi4kq",
                     Collections.emptyList() );
        default -> throw new UsernameNotFoundException( username );
      };
    }
   };
  }
}
```

To make the data types more explicit, an inner anonymous class is used in the implementation. However, a lambda expression could also be used, and we'll explore that later.

Spring calls method `loadUserByUsername(…)` and passes a username. The implementation must look in the "database" to see if the user exists and, if so, build a `UserDetails` object. Otherwise, there is a `UsernameNotFoundException`.

The username is an email address, and two users are recognized as examples, which are also recorded in the database. There are two ways to build the `UserDetails` object. Both ways are shown in the code: one way is through the fluent API, and the other is through the constructor of the `User` class. With the password, an interesting prefix `{noop}` appears that tells Spring Security that the password isn't hashed. We'll come to this in Section 9.17.9.

No Longer 401 Unauthorized

We now have all the information together about how individual paths are secured and users are authenticated by name. Let's see how this would look with the IntelliJ HTTP client:

```
GET localhost:8080/stats
Authorization: Basic fillmore.fat@wyman.co u87szdzwr6j
↩
```

The result would look like this:

```
HTTP/1.1 200
Set-Cookie: JSESSIONID=AD5909EBA19A2D45A5746599A53F1EFF; Path=/; HttpOnly
…
```

The 200 status points out that the registration was successful.

Note

Upon initial inspection, it may seem that the password is transmitted in plain text; however, this isn't the case. Both the headers and body of the transmission are encrypted using Transport Layer Security (TLS) to ensure secure communication between the client and server. This encryption method ensures that sensitive information such as passwords can't be intercepted or read by third-party entities.

UserDetails and User *

Let's take another quick look at Figure 9.20 and the type relationships between `UserDetails` and `User`.

The `User` class has two constructors. The first one is rather simple, and the second one configures everything that a loaded user from a data store can contain in terms of information.

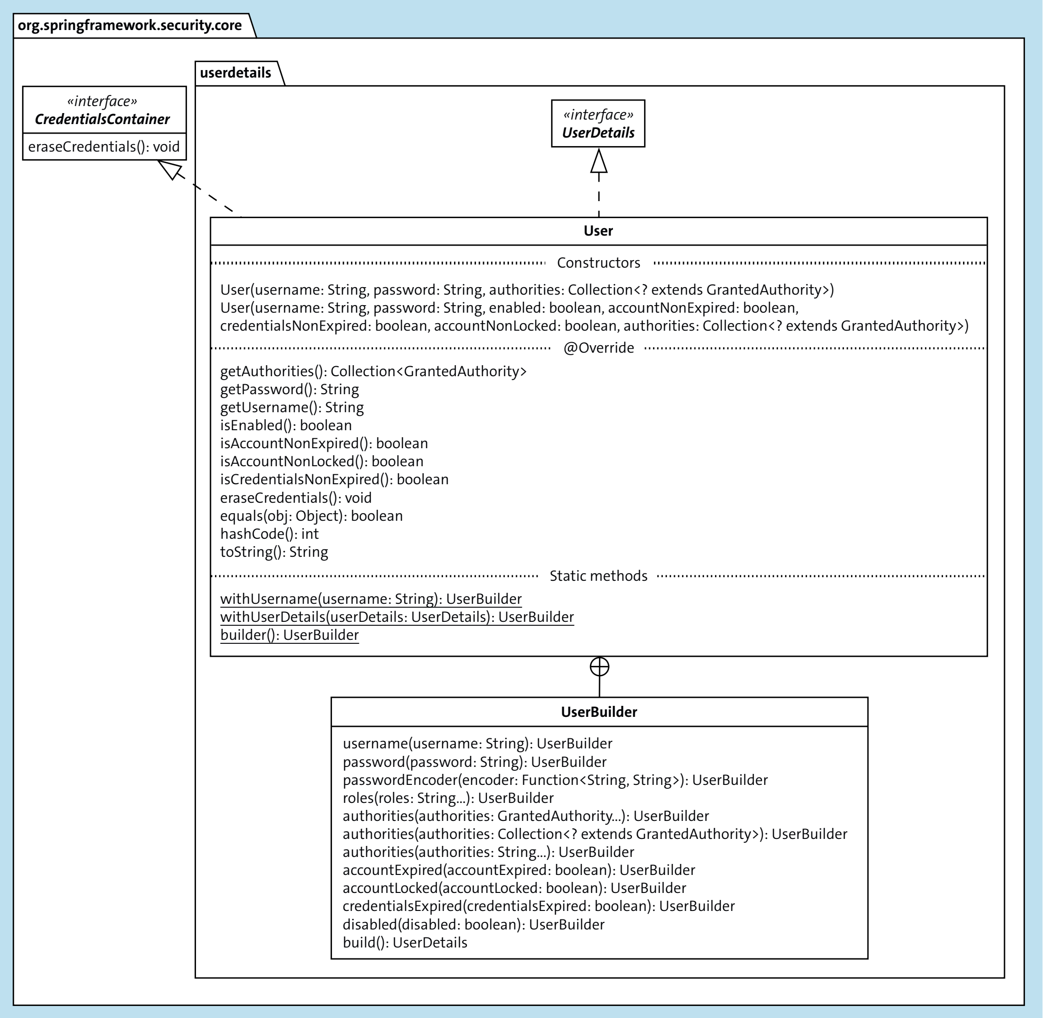

org.springframework.security.core
«interface»
CredentialsContainer
eraseCredentials(): void
userdetails
«interface»
UserDetails
User
Constructors
User(username: String, password: String, authorities: Collection<? extends GrantedAuthority>)
User(username: String, password: String, enabled: boolean, accountNonExpired: boolean,
credentialsNonExpired: boolean, accountNonLocked: boolean, authorities: Collection<? extends GrantedAuthority>)
@Override
getAuthorities(): Collection<GrantedAuthority>
getPassword(): String
getUsername(): String
isEnabled(): boolean
isAccountNonExpired(): boolean
isAccountNonLocked(): boolean
isCredentialsNonExpired(): boolean
eraseCredentials(): void
equals(obj: Object): boolean
hashCode(): int
toString(): String
Static methods
withUsername(username: String): UserBuilder
withUserDetails(userDetails: UserDetails): UserBuilder
builder(): UserBuilder
UserBuilder
username(username: String): UserBuilder
password(password: String): UserBuilder
passwordEncoder(encoder: Function<String, String>): UserBuilder
roles(roles: String...): UserBuilder
authorities(authorities: GrantedAuthority...): UserBuilder
authorities(authorities: Collection<? extends GrantedAuthority>): UserBuilder
authorities(authorities: String...): UserBuilder
accountExpired(accountExpired: boolean): UserBuilder
accountLocked(accountLocked: boolean): UserBuilder
credentialsExpired(credentialsExpired: boolean): UserBuilder
disabled(disabled: boolean): UserBuilder
build(): UserDetails

Figure 9.20 “UserDetails” Interface with the Implementation of the “User” Class

9.17.9 Spring-Managed Bean PasswordEncoder

`UserDetailsService` is responsible for loading the user information, which typically includes a cryptographically hashed password. However, this service doesn’t verify the password for authentication purposes. For this reason, Spring Security provides a separate data type called `PasswordEncoder`, which checks whether the password matches the stored hash. The `UserDetailsService` solely loads the user information and throws an exception if the user isn’t found. If the user is successfully loaded, the `PasswordEncoder` is then used to validate the user’s provided password against the stored hash.

A `PasswordEncoder` declares three methods, as shown in Figure 9.21.

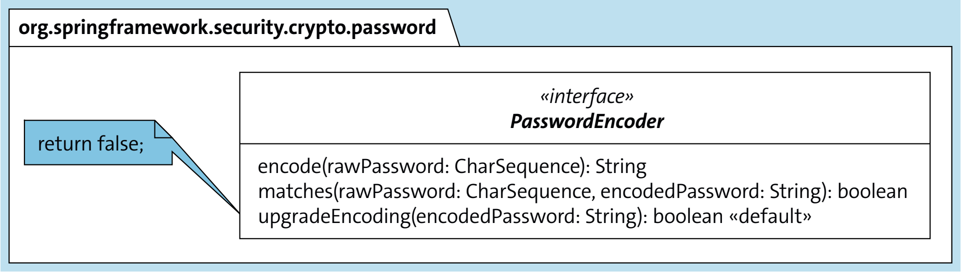

org.springframework.security.crypto.password
return false;
«interface»
PasswordEncoder
encode(rawPassword: CharSequence): String
matches(rawPassword: CharSequence, encodedPassword: String): boolean
upgradeEncoding(encodedPassword: String): boolean «default»

Figure 9.21 "PasswordEncoder" for Matching Passwords

The most important method is `matches(…)`. The first argument passed is the *raw password*. This is what the client sends. It's unencrypted and not hashed. In our case, the password comes directly from the header:

```
Authorization: Basic fillmore.fat@wyman.co u87szdzwr6j
```

The second argument is the (usually hashed) password from the data store. The `PasswordEncoder` implementation compares both and returns `true` if the passwords match and `false` otherwise.

PasswordEncoder and Implementations

The `PasswordEncoder` interface is implemented by different classes. The UML diagram in Figure 9.22 gives an overview.

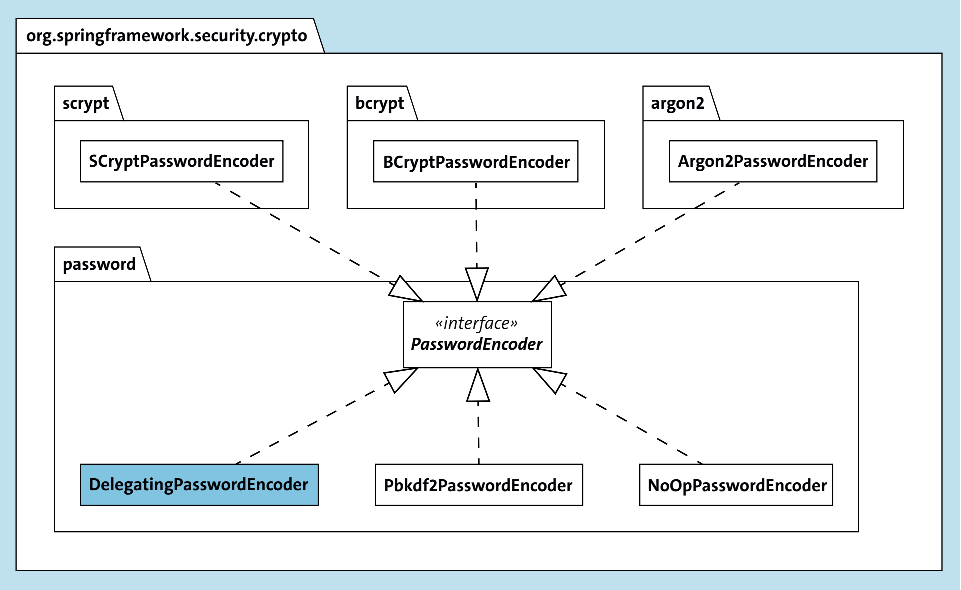

org.springframework.security.crypto
scrypt
SCryptPasswordEncoder
bcrypt
BCryptPasswordEncoder
argon2
Argon2PasswordEncoder
password
«interface»
PasswordEncoder
DelegatingPasswordEncoder
Pbkdf2PasswordEncoder
NoOpPasswordEncoder

Figure 9.22 Different "PasswordEncoder" Implementations

There are concrete `PasswordEncoders` for different hash algorithms, for example, *scrypt*, *bcrypt*, and *Argon2*. Among the classes, there is also `NoOpPasswordEncoder`, which simply compares strings in plain text. Highlighted in the diagram is the `DelegatingPasswordEncoder`, which isn't a password encoder in the true sense, but rather delegates to the concrete password encoder based on a prefix.

DelegatingPasswordEncoder

By default, the `DaoAuthenticationProvider` uses a `DelegatingPasswordEncoder` to match passwords;[88] however, as a Spring-managed bean, no `PasswordEncoder` is registered. The class delegates to another `PasswordEncoder` implementation based on a prefixed identifier. The format is `{id}EncodedPassword`. Internally, the `DelegatingPasswordEncoder` redirects to the specified implementations for the following prefixes:

- `{noop}` → `NoOpPasswordEncoder`
- `{bcrypt}` → `BCryptPasswordEncoder`
- `{pbkdf2}` → `Pbkdf2PasswordEncoder`
- `{scrypt}` → `SCryptPasswordEncoder`
- `{sha256}` → `StandardPasswordEncoder`

We had written the following in a previous program: `new User( username, "{noop} mk8-suwi4kq", Collections.emptyList() )`. That is, `{noop}` selects the `NoOpPasswordEncoder`, and `1234` is the password compared via `equals(…)`:

```
public boolean matches(CharSequence rawPassword,
                       String encodedPassword) {
  return rawPassword.toString().equals(encodedPassword);
}
```

Listing 9.22 https://github.com/spring-projects/spring-security/blob/main/crypto/src/main/java/org/springframework/security/crypto/password/NoOpPasswordEncoder.java, Excerpt

The `DelegatingPasswordEncoder` plays a crucial role in ensuring the security of hashed passwords stored in a database. If the current hashing algorithm becomes compromised, it may become necessary to switch to a different algorithm. However, the software needs to be aware of when this switch should occur. By encoding the algorithm identifier directly, it becomes much simpler to update the hashing algorithm used to secure the passwords in the database.

---

[88] See `PasswordEncoderFactories.createDelegatingPasswordEncoder()` in *https://github.com/spring-projects/spring-security/blob/main/core/src/main/java/org/springframework/security/authentication/dao/DaoAuthenticationProvider.java*.

bcrypt

One popular method is bcrypt, which we mentioned earlier. The strings have a certain structure and can be recognized at first glance. The following strings are hashed versions of Fillmore's password `u87szdzwr6j`:[89]

```
$2a$12$IE5sllLrm.L3GLli.ZBOvevZihVKgXBvrALic8KnXJ9OqVFBOwMtO
$2a$12$pg4L9trk5p7hCCqAqem6l.oOr35UQrTdXYx8L9rflxHvXHnuKS/vW
$2a$12$AppMZTKwev7kUSjd./OfWeABJLnZ.U3DREMl9ygntsMaH1YdFLKOm
$2a$12$aaOuzfZNp/hwhBgy3WOac.k5wEU//cyHICuEKEtfzu3HRR/ochyRC
$2a$12$egNqayqMWkvoLQxa8FV66eykpXzFRdCUTYDk.w6pYi78yFmpa9zpm
```

The hashed password strings always begin with a dollar sign, followed by an algorithm identifier, another dollar sign, and a cost indication reflecting the computational complexity of the hashing algorithm. A higher cost value means that more rounds of the algorithm are performed, making it more difficult for brute force attacks to succeed. The string then includes another dollar sign, a randomly generated value, and the final hashed password. It's important to note that this encoding isn't reversible, meaning that it's impossible to infer the original password from the hash value, such as `u87szdzwr6j`.

For example, if Fillmore has the password `u87szdzwr6j`, the external data store would contain (abbreviated here) `$2a$12$IE…OwMtO`. After loading the `UserDetails`, Spring Web MVC can check whether the password `u87szdzwr6j` matches `$2a$12$IE…OwMtO` via the `BCryptPasswordEncoder`, and it does.

Example

The `encode(…)` method of `PasswordEncoder` creates the encoded password for us:

```
PasswordEncoder encoder1 =
 PasswordEncoderFactories.createDelegatingPasswordEncoder();
System.out.println( encoder1.encode( "u87szdzwr6j" ) );
// e.g. {bcrypt}$2a$10$6Y2XfwOUckGaYqQj3aQ/tuZl2X98rVQjbwiIxfqPVcr4R3YehNZzq
PasswordEncoder encoder2 = new BCryptPasswordEncoder();
System.out.println( encoder2.encode( "u87szdzwr6j" ) );
// e.g. $2a$10$kbOfJyGACamjjRO5PAruuuNtvPOrpeZ1GolosctySlIF6iAvSP3jK
```

The `PasswordEncoderFactories` class provides a preconfigured `PasswordEncoder` that has bcrypt set as default. It also writes the prefix before the encoded string.

User with bcrypt Password

In the data store, only the prefix `{bcrypt}` is prepended because, after all, `DelegatingPasswordEncoder` takes the encoded string. In our own `UserDetailsService`, the string could be changed with `{noop}` to the following:

---

[89] If you want to try it out, visit *https://bcrypt-generator.com*.

```
User
 .withUsername( username )
 .password(
   "{bcrypt}$2a$12$IE5sllLrm.L3GLli.ZBOvevZihVKgXBvrALic8KnXJ9OqVFBOwMtO"
  )
 .roles()
 .build()
```

If the password is to be stored in bcrypt format, a `BCryptPasswordEncoder` can be defined as a `@Bean`, and then the `DelegatingPasswordEncoder` will no longer be used. This can be done as follows:

```
@Bean
public PasswordEncoder passwordEncoder() {
  return new BCryptPasswordEncoder();
}
```

The prefix must then no longer occur.

In the next step, we need to store the passwords only in the bcrypt format:

```
User
 .withUsername( username )
 .password(
   "$2a$12$IE5sllLrm.L3GLli.ZBOvevZihVKgXBvrALic8KnXJ9OqVFBOwMtO"
  )
 .roles()
 .build()
```

9.17.10 BasicAuthenticationFilter

In Spring, incoming requests aren't directly sent to the controller, but are first passed through filters. When dealing with secured paths, the process involves Spring Security reading the Basic Authentication header of an HTTP request from a client. It then loads the user's data and attempts to match the password with a `PasswordEncoder`. If the password is a match, the request is forwarded. If not, the server responds with HTTP status code 401.

`BasicAuthenticationFilter`, which uses `DaoAuthenticationProvider` in the background, plays a crucial role in this process. It's highly recommended to examine the source code of this filter as it's easily comprehensible.[90]

---

[90] *https://github.com/spring-projects/spring-security/blob/main/web/src/main/java/org/springframework/security/web/authentication/www/BasicAuthenticationFilter.java*

9.17.11 Access to the User

Once the filter has authenticated the client, the handler method is called. There, the `Authentication` can be obtained from the `SecurityContextHolder`, the user ID can be fetched from it, and associated data can be loaded from a database, for example. The following example returns the user's "name":

```
// http://localhost:8080/authentication
@GetMapping( "/authentication" )
public String printPrincipalName() {
  Authentication authentication =
      SecurityContextHolder.getContext().getAuthentication();
  String principalName = authentication.getName();
  return principalName;
}
```

There is another possibility—a handler method can directly "wish" for an `Authentication` or `Principal` object:

```
// http://localhost:8080/principal
@GetMapping( "/principal" )
public String currentUserName( Authentication principal ) {
// public String currentUserName( Principal principal ) {
  return principal.getName();
}
```

However, this variant only works if authentication is successful; otherwise, there is an exception. If the web service should also work when a user isn't logged in, then you're better off with the first variant because then the `Authentication` reference is simply `null`.

Example: Load Real Unicorns

We previously built our `UserDetailsService` with two users hard-coded. We want to change that to really read users from the database. To complete this, we can use a `UnicornRepository` because the username (i.e., the email address) and the password are stored in the database, and the repository gives us the access. Let's assume that the `UnicornRepository` offers method `findByEmail(…)`. Then, a functional implementation of `UserDetailsService` with database access can look like this:

```
@Configuration( proxyBeanMethods = false )
public class UserDetailsServiceConfig {

 @Bean
 UserDetailsService userDetailsService( UnicornRepository unicorns ) {
  return email ->
```

```
      unicorns.findByEmail( email )
              .map( unicorn -> User.withUsername( unicorn.getEmail() )
                                   .password( unicorn.getPassword() )
                                   .roles( "USER" ).build() )
              .orElseThrow( () -> new UsernameNotFoundException(email) );
  }
}
```

The lambda expression makes the code even a little more compact. The role is simply a string and can also be checked later. That is a task for authorization.

This brings us full circle. We can program REST endpoints, secure paths, and ensure that only those unicorns who validly log in can access the secured paths.

9.17.12 Authorization and the Cookie

The process of authentication and subsequent communication between an HTTP client and server involves the exchange of several headers and cookies. First, when a client logs into an HTTP server, an `Authorization` header is set, which contains the user's login credentials.

```
Authorization: Basic ABCABCABCABCABCABCABC
```

The server then attempts to authenticate the user and, if successful, responds with a `Set-Cookie` header in the response, which contains a session ID:

```
Set-Cookie: JSESSIONID=1234567890; Path=/; HttpOnly
```

The session ID acts as an identifier for the user's session and is used by the server to keep track of the user's stateful communication. The `Path=/; HttpOnly` option in the `Set-Cookie` header specifies that the cookie shouldn't be accessible via JavaScript.[91]

The unique feature of this communication is that subsequent requests from the client to the server will automatically include the session ID cookie in the `Cookie` header:

```
GET localhost:8080/stats
Cookie: JSESSIONID=1234567890
```

This happens seamlessly in web browsers and requires no intervention from the user, which means that once the server has authenticated the client and issued a session ID cookie, subsequent requests from the client will be associated with the same session ID. This ensures that the server can maintain the client's state and provide personalized services. So, there is a switch to stateful communication because the moment we exchange a cookie and it's state, the server knows the client. If the cookie were taken away from the server, then it wouldn't know that the client had authenticated earlier.

---

[91] *https://owasp.org/www-community/HttpOnly* explains why this makes web applications more secure.

However, when programming communication via a REST API in Java, developers need to manually implement this automatic exchange of cookies. They need to handle the creation and sending of cookies in the response header and the storage and sending of cookies in the request header for subsequent communications.

9.17.13 Token-Based Authentication with JSON Web Tokens

Token-based authentication provides an alternative to the traditional method of using a username and password, as well as the use of a cookie ID for session management. With token-based authentication, a token is generated and sent with every request, much like a cookie value. However, there is a significant difference between the two approaches.

In token-based authentication, the server doesn't need to store the token. Instead, it accepts the token and verifies its validity before granting access to the user. This is different from cookie-based authentication, where the server needs to maintain a record of the session ID cookie to identify the user's session.

Token-based authentication is commonly used in *single sign-on* systems, where a user logs in once with their username and password and receives a token. This token can then be used to access other servers without the need for further verification, as the server trusts the token issuer. This provides a seamless experience for the user, as they can access multiple services without having to repeatedly enter their login credentials.

JSON Web Tokens

Of course, a token could be in any format. But there is a standard that is supported by libraries called *JSON Web Token* (JWT; *www.rfc-editor.org/rfc/rfc7519*).

The JWT website (*https://jwt.io/*) gives an example of a JWT:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM ↩
0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM ↩
5MDIyfQ.SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c
```

A JWT is Base64 encoded, and there are three parts separated by dots. Decoded it looks like this:

```
{"alg":"HS256","typ":"JWT"}{"sub":"1234567890","name":"John Doe","iat":151623902
2} pDx ?Lx"UP
```

You can see two small JSON documents, and that's where the name JWT comes from: *JSON* Web Token. The last part is binary, not JSON.

Let's take a closer look at these three parts.

- `{"alg": "HS256", "type": "JWT"}`
  The first part is the header, also called the *JOSE header*. This header contains two pieces of information: an algorithm is specified, and the type is `JWT`.
- `{"sub": "1234567890", "name": "John Doe", "iat":1516239022}`
  Next comes an object with data (*payload*), that is, the *claims*. Some of these keys in the JSON document are standardized, but you can also assign arbitrary free keys and set arbitrary properties there.
- **Signature bytes**
  The third part is the signature. The signature ensures that the client sending the JWT doesn't simply change the token. Without a digital signature, the client could include arbitrary information and pretend to have an entirely different identity. However, this can't happen because the client can't build its own token. The client gets a token that was built by someone else and can forward it. Any modification would always be noticed.

Let's stay briefly with the topics of signature and algorithm.

Signing of JSON Web Tokens

Two different methods are recommended for signing. The two algorithms are called RS256 and ES256 for short:

- **RSA (named after its developers, Rivest, Shamir, and Adleman)**
  RSA is an asymmetric cryptographic technique. It uses a key pair consisting of a public key (for encrypting or verifying signatures) and a private key (for decrypting or signing data).
- **ES256 (a Hash-based Message Authentication Code [HMAC])**
  HMAC is a cryptographic hash function with a secret key. The cryptographic hash function SHA-256 (*Secure Hash Algorithm*) and the *Elliptic Curve Digital Signature Algorithm* (ECDSA) are used.

The header and payload data are signed using either HMAC-SHA-256 or RSA-SHA-256. At the end, they are Base64 encoded—and this is the third and last step to generate the JWT.

Process for Token-Based Authentication

Token-based authentication involves obtaining a token in the first step, which can then be used for subsequent operations by the client. The process of obtaining a token starts with the client authenticating itself. This involves the client contacting the server with the user's login credentials, such as their username and password or other identification features. The following is an example of this process:

```
POST http://localhost:8080/login
Authorization: Basic fillmore.fat@wyman.co u87szdzwr6j
```

In the code examples, the request is depicted using the notation of the IntelliJ HTTP client. The username and password are Base64 encoded and are included either in an `Authorization` header or in the request body. However, if the credentials are sent via the header, Spring can automatically parse and evaluate them.

Once the request reaches the server, the credentials are verified. If the username and password are valid, a JWT is returned. The token may resemble the following:

```
HTTP/1.1 200
[… Header …]
Keep-Alive: timeout=60
Connection: keep-alive

eyJraWQiOiIwNWNlMGU4ZC01MDc3LTQ0YTMtOWM0NiO0MGI4NjQzN2Y2Zj ↩
ciLCJhbGciOiJSUzI1NiJ9.eyJzdWIiOiJmaWxsbW9yZSIsImV4cCI6MTY ↩
2NzMyNDc2NCwiaWF0IjoxNjY3MzIxMTY0fQ.NTSE7bvAOE2Cyretcro5ij ↩
POqXGswuvxmaki7TYUHL_6OqU6qm44eY_L3r5cbdJxniHaoqOkiBezIwxc ↩
Npru8X7GlYuT74N6w4T8iyFEurdx1qgwtoBBSHOQQ8hULK-y2n6tnb2l2l ↩
Iiv8LC-rkgjhlkbAOqQL3GMvHtL3E6AAAIGVqNNCb_IUIi7lTgpXEBhO4J ↩
mNAljPdFkWM5T4grpx_oMBO1UP4VgpPwGwx72IVb3VtyQSZe3BwQEJBRL- ↩
UhlDGIMgBzoq8VajO9_pGavBfiyz8o1qXwLqrDPYIVWCto5cKF864V6im4 ↩
zerbovXIj9OuFNr7M8-pNI3wuK67aA
```

In the first step of the communication, the client doesn't call any productive web service endpoints, but only requests the token.

In each of the following steps, the client will send the token to the server, thereby informing the server that is entitled to communicate. A `GET` request could look like this:

```
GET http://localhost:8080/
Authorization: Bearer
eyJraWQiOiIwNWNlMGU4ZC01MDc3LTQ0YTMtOWM0NiO0MGI4NjQzN2Y2Zj ↩
ciLCJhbGciOiJSUzI1NiJ9.eyJzdWIiOiJmaWxsbW9yZSIsImV4cCI6MTY ↩
2NzMyNDc2NCwiaWF0IjoxNjY3MzIxMTY0fQ.NTSE7bvAOE2Cyretcro5ij ↩
POqXGswuvxmaki7TYUHL_6OqU6qm44eY_L3r5cbdJxniHaoqOkiBezIwxc ↩
Npru8X7GlYuT74N6w4T8iyFEurdx1qgwtoBBSHOQQ8hULK-y2n6tnb2l2l ↩
Iiv8LC-rkgjhlkbAOqQL3GMvHtL3E6AAAIGVqNNCb_IUIi7lTgpXEBhO4J ↩
mNAljPdFkWM5T4grpx_oMBO1UP4VgpPwGwx72IVb3VtyQSZe3BwQEJBRL- ↩
UhlDGIMgBzoq8VajO9_pGavBfiyz8o1qXwLqrDPYIVWCto5cKF864V6im4 ↩
zerbovXIj9OuFNr7M8-pNI3wuK67aA
```

The client sets `Authorization: Bearer + JWT` in the header.

After the client sends the data to the server, the server will verify the validity of the JWT. If the token is valid, the handler method can then generate the appropriate response.

Note

A token should never be given out casually or without proper authorization measures in place. This is because anyone who gains access to a token can potentially access the corresponding endpoint or functionality associated with that token. A token doesn't necessarily represent an individual user, but rather represents a set of permissions or rights to perform certain actions. Therefore, it's essential to carefully consider who is granted access to a token and what level of access that token provides. To ensure proper security, tokens are often associated with a set of roles or permissions that dictate the actions that can be performed using that token. Additionally, tokens are often designed to have a limited lifespan or expiration date to help prevent unauthorized access in case a token is lost or stolen.

Pseudocode for the Two Steps

To implement token-based authentication in a Spring application, we can follow these two general steps. To start, let's take a look at some pseudocode to understand what Spring requires for this process.

In both steps, it's necessary to encode and decode the JWT. To encode, we need to construct a JWT, and to decode, we need to reconstruct it. To accomplish this, we can use helper types that we'll call `JwtEncoder` and `JwtDecoder`.

Assuming we have access to these types, we can proceed with the following steps:

```
@Autowired JwtEncoder jwtEncoder;
@Autowired JwtDecoder jwtDecoder;
```

The exchange starts with the server supplying a JWT. An encoder has the task to bring a JWT into the Base64 encoding. In principle, it looks like this:

```
@PostMapping( "/login" )
public String token( Authentication authentication ) {
  return jwtEncoder.encode( token_data ).getTokenValue();
}
```

Usually one uses a `@PostMapping`. The user must log in with a username and password, and if the credentials are correct, the response is the JWT.

When a client makes a call using a JWT, the token is sent back to the server. To extract the JWT, the string `bearer` must be removed from the `Authorization` header. After decoding the token, the validity of the object must be confirmed, and data such as the user ID can be read from the JSON object. Here is an example in pseudocode:

```
@*Mapping( secured-path )
public String home( @Header("Authorization") String token ) {
  jwtDecoder.decode( token - "Bearer " );
  …
}
```

What we've shown here schematically isn't far from what we actually have to realize in Spring.

Jwt Encoder and Jwt Decoder

The data types `JwtEncoder`[92] and `JwtDecoder`[93] actually exist. They are two interfaces from Spring Security—more precisely, from package `org.springframework.security.oauth2.jwt`. These interfaces are implemented by the `NimbusJwtEncoder` and `NimbusJwtDecoder` classes. The two classes help to build or parse a JWT.

Before the objects are usable, they have to be configured because a token is always signed. We have two options for this: HMAC or RSA. We have to choose a secret for symmetric encryption, a key, and have a key pair if using RSA. Both methods are fine. Because it's a bit easier, we use the RSA encryption.

The `JwtDecoder` and the `JwtEncoder` must then be connected to the key. Before we can do that, we must have an RSA key pair. We can read this in or generate it ourselves.

com.nimbusds.jose.jwk.RSAKey

The *Java SE library* can generate an RSA key pair. However, Spring Security wants a different data type and not a pure RSA key that comes from Java SE. Spring Security internally falls back on types from the open-source library *Nimbus* (*https://bitbucket.org/connect2id/nimbus-jose-jwt/*), so it's no wonder that the interface implementations `JwtDecoder` and `JwtEncoder` are then called `NimbusJwtDecoder` and `NimbusJwtEncoder`.

The following code snippet puts an RSA key into context as a bean:

```
@Bean
RSAKey rsaKey() throws NoSuchAlgorithmException {
  var generator = KeyPairGenerator.getInstance( "RSA" );
  generator.initialize( 2048 );
  var keyPair = generator.generateKeyPair();
  return new RSAKey.Builder( (RSAPublicKey) keyPair.getPublic() )
           .privateKey( keyPair.getPrivate() )
```

---

[92] *https://docs.spring.io/spring-security/site/docs/current/api/org/springframework/security/oauth2/jwt/JwtEncoder.html*

[93] *https://docs.spring.io/spring-security/site/docs/current/api/org/springframework/security/oauth2/jwt/JwtDecoder.html*

```
            .keyID( UUID.randomUUID().toString() )
            .build();
}
```

The `RSAKey` doesn't have to be in context as a bean, but, this way, we can later use the `RSAKey` for initializing the `NimbusJwtDecoder` and the `NimbusJwtEncoder`.

JwtEncoder

The RSA object can be used to initialize `NimbusJwtEncoder`:

```
@Bean
JwtEncoder jwtEncoder( RSAKey rsaKey ) {
  return new NimbusJwtEncoder(
    new ImmutableJWKSet<>( new JWKSet( rsaKey ) )
  );
}
```

The `NimbusJwtEncoder` class implements `JwtEncoder`. In this context, we wouldn't need to set the object either, but, this way, it's easily accessible for the next step.

Login Controller

The next step is the point where the `JwtEncoder` builds a JWT. This could be a separate controller that looks like this:

```
@RestController
class AuthController {

  private final JwtEncoder jwtEncoder;

  AuthController( JwtEncoder jwtEncoder ) {
    this.jwtEncoder = jwtEncoder;
  }

  @PostMapping( "/login" )
  public String token( Authentication authentication ) {
    Instant now = Instant.now();
    JwtClaimsSet claims = JwtClaimsSet.builder()
        .issuedAt( now )
        .expiresAt( now.plus( 1, ChronoUnit.HOURS ) )
        .subject( authentication.getName() )
        .build();
    return jwtEncoder.encode( JwtEncoderParameters.from( claims ) )
```

```
                     .getTokenValue();
  }
}
```

The `AuthController` class gets the `JwtEncoder` injected and can use it to build a JWT as we wish. The `JwtEncoder` bean wouldn't have to be in context for this, but that is convenient for the example.

In the login handle method, we obtain the `Authentication` object as the user needs to authenticate for the first time. The `Authentication` object is useful because it allows us to access the username and permissions, which can be included in the JWT.

When you want to build a JWT object, claims are set that make up the payload. The claims are set via the Fluent API and later passed to the `JwtEncoder`. The generated string can be returned by the controller or alternatively put into a JSON object; the return format isn't relevant.

Next, we turn to the controller, which receives the JWT and must validate it. Only when the object is valid can the handler methods be called.

SecurityFilterChain and oauth2ResourceServer

We don't have to do the web security token checking ourselves because Spring Security does that. We just need to add a line to the `HttpSecurity` filter chain:

```
@Bean
SecurityFilterChain securityFilterChain( HttpSecurity http )
    throws Exception {

 return http
         .csrf( csrf -> csrf.disable() )
         .authorizeRequests( … )
         .oauth2ResourceServer( OAuth2ResourceServerConfigurer::jwt )
         …
         .build();
}
```

Filters are a central concept of Spring Web. Through `oauth2ResourceServer(Customizer<OAuth2ResourceServerConfigurer<HttpSecurity>>)`, a `BearerTokenAuthenticationFilter` (a `OncePerRequestFilter`) is appended to the filter chain. The `BearerTokenAuthenticationFilter` internally validates the token.

When we wrote `AuthController`, we accessed the `JwtEncoder` object ourselves, and `JwtEncoder` was really needed as a bean. With the `JwtDecoder`, it's a little different because the `OAuth2ResourceServerConfigurer` needs a `JwtDecoder`. Because a `JwtDecoder` is associated with a key such as `JwtEncoder`, we need to provide the bean.

JwtDecoder

As an implementation of `JwtDecoder`, we got to know `NimbusJwtDecoder`. Furthermore, the `NimbusJwtDecoder` is associated with the `RSAKey`:

```
@Bean
JwtDecoder jwtDecoder( RSAKey rsaKey ) throws JOSEException {
  return NimbusJwtDecoder.withPublicKey( rsaKey.toRSAPublicKey() )
                         .build();
}
```

The structure of the `NimbusJwtDecoder` is simple: it gets the public key, and that is enough for the filter.

To summarize, in token-based authentication, the client sends a JWT to the server with each request. The server is stateless and doesn't store the token. Therefore, it doesn't need to authenticate the client every time, but rather trusts that the token is valid and issued by a trusted entity. This approach allows for more efficient and scalable authentication without the need for the server to maintain a session or store user credentials.

Session-Based versus Token-Based Authentication

When it comes to authentication, there are two common options: using `Authorization` headers with later cookie exchange or relying on token-based authentication. Both approaches have their advantages and disadvantages.

One major issue with using cookies for authentication is that they are vulnerable to *Cross Site Request Forgery* (CSRF).[94] This means that an attacker can impersonate a user and execute unauthorized actions on their behalf.

Additionally, cookies pose scalability challenges. This is because cookies are stored on both the client side, that is, the browser, and the server side. While storage on the client side is usually not an issue, server-side storage can become a problem when dealing with large amounts of data per client. For example, if hundreds of thousands of clients log in, the server has to store their cookies and associated data, which can become quite heavy.

Another scalability challenge with cookies arises when a client communicates with multiple servers in a server farm. In this scenario, the server with which the client first communicates stores the cookie. However, if the next request from the client goes to a different server in the farm, that server won't know the client and will fail the operation. To overcome this, servers must exchange cookie data with each other. However, this state management process is costly and must be considered when scaling the system.

Cookies are restricted to a specific domain or subdomain, and they aren't sent to any other domain. This can pose a problem in scenarios where multiple servers need to form a federation, but have different domain names. In such cases, the cookie can't be shared among the servers, which can result in reduced functionality and efficiency.

---

[94] *https://owasp.org/www-community/attacks/csrf*

Tokens are a popular authentication method in a microservice environment because they are stateless and highly scalable. Once a user is authenticated, the token can be passed to any entity that trusts it, eliminating the need for constant queries to the server to check the user's validity.

However, tokens also have a major disadvantage: a compromised token can be used by malicious clients, especially if the token has a long lifespan. Because tokens are stateless and not stored on the server, it's almost impossible to reverse them or track their use. If a token becomes invalid, the server has no way of knowing this unless it maintains a list of invalid tokens, which can become cumbersome.

Therefore, it's crucial to take special care to ensure that tokens don't fall into the wrong hands. In comparison, cookies are easier to manage because a user can log out, which deletes the cookie and closes the session. If an attacker gains access to the cookie, they must log in again with the correct credentials. To mitigate the risk of compromised tokens, it's a best practice to set a lifetime that isn't too long.

9.18 Consume RESTful Web Services

So far, we've implemented web services only on the server side. In this section, we want to move to the other side and become a client, that is, a caller.

9.18.1 Classes for Addressing HTTP Endpoints

When it comes to interacting with RESTful web services as a client, there are different options available. In Java SE, you could make use of the `URL` and `URLConnection` classes, which have been present since Java 1.0. However, these classes have some limitations and aren't very user-friendly. With the release of Java 11, the situation has improved as the `java.net.http` package was introduced. This package provides a `HttpClient` class along with classes such as `HttpResponse` and `HttpRequest`. While these classes make it easier to work with RESTful web services, configuring them can still require a significant amount of effort to successfully make a web service call.

Network calls are even easier with Spring data types. For this, there is the `RestTemplate` on the one hand and the `WebClient` from the *Spring Reactive* universe on the other. Under Spring Framework 6.1, the `RestClient` class will most likely be added, which is very related to `WebClient` in terms of API, but does not use reactive data types.[95]

Note: From RestTemplate to WebClient

The `RestTemplate` class is still in the library and isn't deprecated, but the Java documentation clearly states that this class is in maintenance mode, and the `WebClient` data type should be used. The `WebClient` has a more modern API and also supports the asynchronous reactive mode besides the synchronous mode.

---

[95] *https://spring.io/blog/2023/07/13/new-in-spring-6-1-restclient*

All named classes enable network calls regardless of the type of API, be it RESTful, SOAP, GraphQL, or even an HTML page from Wikipedia's server.

An alternative approach is to create declarative HTTP clients. To explore this approach, we'll start by examining the `WebClient` class before discussing declarative HTTP clients, leaving the Java SE classes aside for now.

9.18.2 A WebClient Example

Type `WebClient`[96] comes from the field of reactive programming, as can be seen from its package name: `org.springframework.web.reactive.function.client`. If you want to use the class, include the following dependency:

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-webflux</artifactId>
</dependency>
```

As a dependency, the client doesn't exist separately, so the whole package is necessary. This is unfavorable because you get more than just the `WebClient` class with the Spring Boot launcher WebFlux—this is the sour apple you have to bite into.

The actual call is short, with four lines:

```
WebClient webClient = WebClient.create( "https://httpbin.org/get" );
Mono<String> result = webClient.get().retrieve().bodyToMono( String.class );
String body = result.block();
System.out.println( body );
```

Data type `Mono` originates from the context of reactive programming. The website *https://httpbin.org/get* returns a body, which we accept as a string. We wait for the result using the `block()` method and output it.

Random Data of Individuals

We can use the `WebClient` for a small example. *https://randomuser.me/* is a web page that can generate random users: with a random name, random gender, random date of birth, and so on (see Figure 9.23).

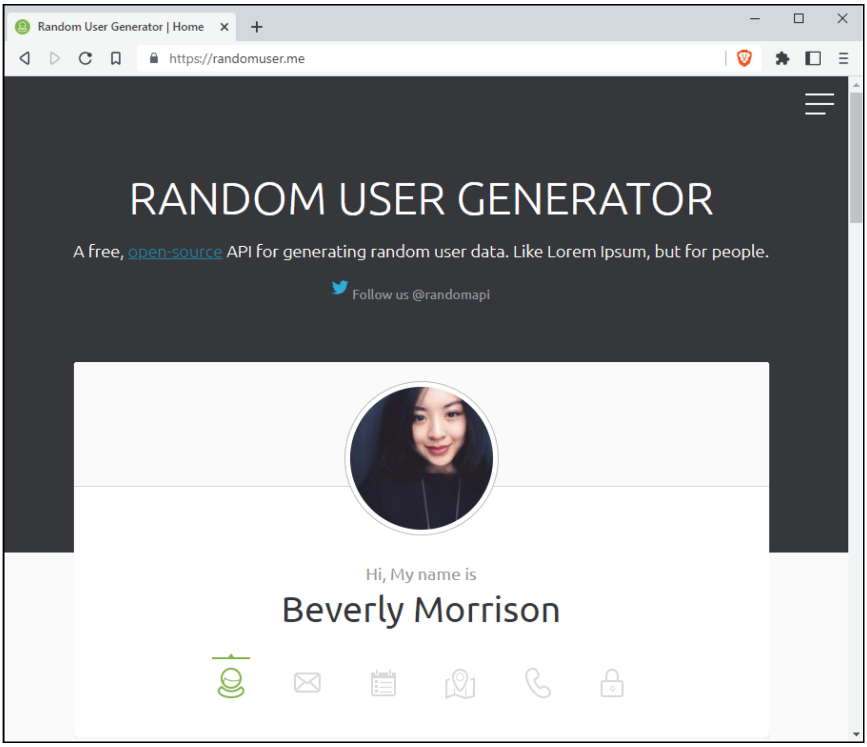
The service can also be accessed through a straightforward web service at *https://randomuser.me/api/*. The response you'll receive is a JSON document that can be visualized as follows:

```
{
  "results": [
    {
```

---

[96] *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/reactive/function/client/WebClient.html*

```
      "gender": "male",
      "name": {
        "title": "Mr",
        "first": "Vilhelm",
        "last": "Farestveit"
      },
      "location": {
        "street": {
          "number": 2594,
          "name": "Vetlandsfaret"
        },
        "city": "Notodden",
        "state": "Hordaland",
        "country": "Norway",
        "postcode": "2333",
        "coordinates": {
          "latitude": "42.9654",
          "longitude": "-83.2432"
        },
        …
}
```



RANDOM USER GENERATOR

A free, open-source API for generating random user data. Like Lorem Ipsum, but for people.

Follow us @randomapi

Hi, My name is

Beverly Morrison

Figure 9.23 Random Person from the Random User Generator

So, how could we address this endpoint with the `WebClient`? We know the URL, and the beginning would be as follows:

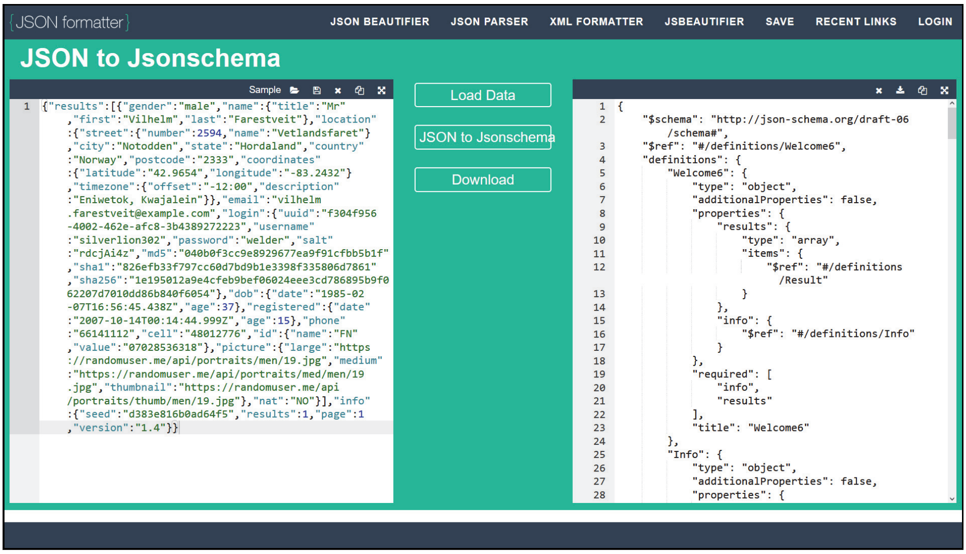
```
WebClient webClient = WebClient.create( "https://randomuser.me/api/" );
Mono<XXX> result = webClient.get().retrieve().bodyToMono( XXX.class );
XXX body = result.block();
System.out.println( body ); // body.results.get( 0 ).name.first
```

The initial expressions are straightforward, but subsequently, we must specify a data type represented by the symbol `XXX`. However, the source of this data type remains unclear, and we would rather not construct a `String`. Instead, we need a Java object built from the JSON object, but we lack the necessary container. Previously, when we operated on the server side, we began with a container that Jackson generated a JSON document from. However, as receivers, we don't have a shared codebase with the server, and we're uncertain if the server even uses Java.

Various methods exist to obtain the container class. One option is to manually create a record or class. Nevertheless, this approach is arduous, and we would have to modify the container manually if the format changes. A much superior alternative is to generate the container using a specification, such as a JSON schema or an OpenAPI document.

Convert JSON to a JSON Schema

Not every RESTful web service provides an OpenAPI document, and, unfortunately, the *Random User Generator API* doesn't provide such a document either. Even a JSON schema for the response doesn't exist. In this case, we can make do with having a JSON schema generated from a JSON object. There are several sites that can do this, such as *https://jsonformatter.org/json-to-jsonschema* (see Figure 9.24).



```
{"results":[{"gender":"male","name":{"title":"Mr"
    ,"first":"Vilhelm","last":"Farestveit"},"location"
    :{"street":{"number":2594,"name":"Vetlandsfaret"}
    ,"city":"Notodden","state":"Hordaland","country"
    :"Norway","postcode":"2333","coordinates"
    :{"latitude":"42.9654","longitude":"-83.2432"}
    ,"timezone":{"offset":"-12:00","description"
    :"Eniwetok, Kwajalein"}},"email":"vilhelm
    .farestveit@example.com","login":{"uuid":"f304f956
    -4002-462e-afc8-3b4389272223","username"
    :"silverlion302","password":"welder","salt"
    :"rdcjAi4z","md5":"040b0f3cc9e8929677ea9f91cfbb5b1f"
    ,"sha1":"826efb33f797cc60d7bd9b1e3398f335806d7861"
    ,"sha256":"1e195012a9e4cfeb9bef06024eee3cd786895b9f0
    62207d7010dd86b840f6054"},"dob":{"date":"1985-02
    -07T16:56:45.438Z","age":37},"registered":{"date"
    :"2007-10-14T00:14:44.999Z","age":15},"phone"
    :"66141112","cell":"48012776","id":{"name":"FN"
    ,"value":"07028536318"},"picture":{"large":"https
    ://randomuser.me/api/portraits/men/19.jpg","medium"
    :"https://randomuser.me/api/portraits/med/men/19
    .jpg","thumbnail":"https://randomuser.me/api
    /portraits/thumb/men/19.jpg"},"nat":"NO"}],"info"
    :{"seed":"d383e816b0ad64f5","results":1,"page":1
    ,"version":"1.4"}}
```

```
{
    "$schema": "http://json-schema.org/draft-06
        /schema#",
    "$ref": "#/definitions/Welcome6",
    "definitions": {
        "Welcome6": {
            "type": "object",
            "additionalProperties": false,
            "properties": {
                "results": {
                    "type": "array",
                    "items": {
                        "$ref": "#/definitions
                          /Result"
                    }
                },
                "info": {
                    "$ref": "#/definitions/Info"
                }
            },
            "required": [
                "info",
                "results"
            ],
            "title": "Welcome6"
        },
        "Info": {
            "type": "object",
            "additionalProperties": false,
            "properties": {
```

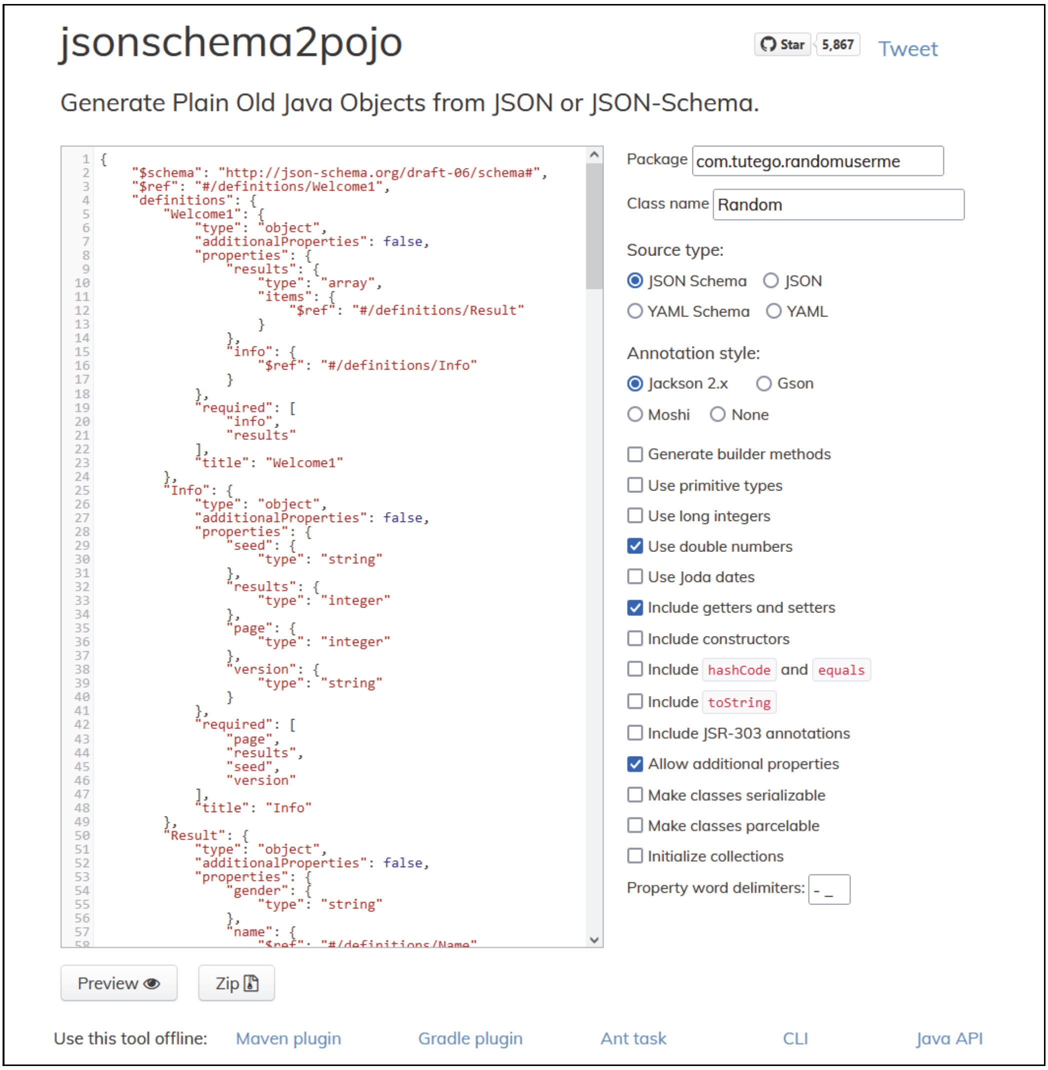
Figure 9.24 JSON Formatter Generating a JSON Schema from a JSON Document

Because a generator can't always recognize the correct data types or repetitions, the generated schema must be considered only as a starting point, not as a finished product.

If the schema description is available, Java containers can be built automatically.

jsonschema2pojo

Generating the Java containers from a JSON schema can be done in two ways. The first way is manually via the *www.jsonschema2pojo.org/* page (see Figure 9.25).



```
{
    "$schema": "http://json-schema.org/draft-06/schema#",
    "$ref": "#/definitions/Welcome1",
    "definitions": {
        "Welcome1": {
            "type": "object",
            "additionalProperties": false,
            "properties": {
                "results": {
                    "type": "array",
                    "items": {
                        "$ref": "#/definitions/Result"
                    }
                },
                "info": {
                    "$ref": "#/definitions/Info"
                }
            },
            "required": [
                "info",
                "results"
            ],
            "title": "Welcome1"
        },
        "Info": {
            "type": "object",
            "additionalProperties": false,
            "properties": {
                "seed": {
                    "type": "string"
                },
                "results": {
                    "type": "integer"
                },
                "page": {
                    "type": "integer"
                },
                "version": {
                    "type": "string"
                }
            },
            "required": [
                "page",
                "results",
                "seed",
                "version"
            ],
            "title": "Info"
        },
        "Result": {
            "type": "object",
            "additionalProperties": false,
            "properties": {
                "gender": {
                    "type": "string"
                },
                "name": {
                    "$ref": "#/definitions/Name"
```

Package: com.tutego.randomuserme

Class name: Random

Source type:

● JSON Schema ○ JSON

○ YAML Schema ○ YAML

Annotation style:

● Jackson 2.x ○ Gson

○ Moshi ○ None

☐ Generate builder methods

☐ Use primitive types

☐ Use long integers

☑ Use double numbers

☐ Use Joda dates

☑ Include getters and setters

☐ Include constructors

☐ Include `hashCode` and `equals`

☐ Include `toString`

☐ Include JSR-303 annotations

☑ Allow additional properties

☐ Make classes serializable

☐ Make classes parcelable

☐ Initialize collections

Property word delimiters: - _

Figure 9.25 Creating Java Containers from a JSON Schema

On the left, you insert the JSON schema into the text field, and on the right, there are settings such as package name, root class, and so on. In principle, you could also put JSON directly on the left, but there should always be a JSON schema for substantial services.

If you click on **Preview**, you can have a look at the containers. Via **Zip**, you download a composite of all Java types as an archive.

Manually inputting data into a text field and then copying the result into a project isn't a viable long-term solution. To circumvent this issue, we can use a Maven plugin. One such plugin is available through the *http://jsonschema2pojo.org*, which provides a graphical interface for an underlying library that can also be accessed via Maven. This project is open source and can be found at *https://github.com/joelittlejohn/jsonschema2pojo*.

To use the plugin, we include the following code in the POM file:

```
<build>
 <plugins>
  <!-- org.springframework.boot:spring-boot-maven-plugin -->
  <plugin>
    <groupId>org.jsonschema2pojo</groupId>
    <artifactId>jsonschema2pojo-maven-plugin</artifactId>
    <executions>
     <execution>
      <configuration>
       <includeGetters>false</includeGetters>
       <includeSetters>false</includeSetters>
       <includeAdditionalProperties>false</includeAdditionalProperties>
       <sourcePaths>
        <sourcePath>
         ${project.basedir}/src/main/resources/randomuserme.jsd.json
        </sourcePath>
       </sourcePaths>
       <targetPackage>com.tutego.randomuserme.dto</targetPackage>
      </configuration>
      <goals><goal>generate</goal></goals>
     </execution>
    </executions>
  </plugin>
 </plugins>
</build>
```

The settings on the *http://jsonschema2pojo.org* website are also found in the plugin. The web page at *https://joelittlejohn.github.io/jsonschema2pojo/site/1.2.1/generate-mojo.html* describes which options are available. The naming of the JSON schema description and the Java package in which the sources come is elementary.

Then, when a build is triggered, the containers are generated automatically. This is convenient because if something changes in the description, there will probably be compiler errors in the client that will be noticed immediately. You should store the JSON schema files centrally.

We can use these containers. In the previous example, where we saw `XXX` before, we would now have the generated container type.

9.18.3 Declarative Web Service Clients

The programmed approach is strongly tied to HTTP, but for the client, it doesn't matter that a RESTful web service is being called in the background. However, the client must correctly set the parameters for the transfers because the framework doesn't take care of this work for us.

Web services can be seen as functions that map an input to an output. So why not model web services as methods that can be called directly? In this case, the method accepts arguments and sends them to the server, which then returns the response. This is the basic idea behind declarative web service clients. In code form, this could look like the following:

```
public interface Profiles {
  @GET( "/api/profiles" )
  List<Profile> findAll();
}
```

For declarative web service clients, you write an interface such as `Profiles` and annotate methods. At runtime, you get an object of this type that handles the server communication. If you've been around Java Enterprise development for a while, you'll be familiar with similar interfaces from Remote Method Invocation (RMI) and SOAP-based web services.

The `GET` annotation is fictitious for the example. We'll get more specific in the following sections.

Libraries for Declarative Web Service Clients

There are several libraries that generate implementations from the interfaces.

- **(Open)Feign (*https://github.com/OpenFeign/feign*)**
  One of the most well-known libraries is Feign. With an interface, a method call is "pretended," but it's actually remote, not local. Originally developed by Netflix, Feign was later abandoned by the company and donated to the open-source community, where it continues to exist as *OpenFeign*. OpenFeign is tightly integrated with Spring, and there is even a special project called *Spring Cloud OpenFeign* that integrates other services, including service discovery (i.e., determining which server to address), failover, and load balancing—everything you would need in a microservices architecture. The

only drawback is that OpenFeign doesn't support reactive clients, and while there are several forks of the project, reactive solutions remain uncommon.

- **Retrofit (*https://square.github.io/retrofit/*)**
  Retrofit is popular in the Android environment. There is an integration into the Spring universe via community projects.
- **Avaje http client (*https://avaje.io/http-client/*)**
  Unlike OpenFeign and Retrofit, the Avaje HTTP client doesn't generate an implementation at runtime via a proxy, but generates a class in the build process. In addition, the project consistently relies on Java 11's HTTP client, so no other dependencies are necessary. With OpenFeign, the HTTP library can be selected, and there are many supported projects.
- **HTTP interface (*https://docs.spring.io/spring-framework/docs/current/reference/html/integration.html#rest-http-interface*)**
  Because the Spring team doesn't control the further development of OpenFeign (it's a separate project), and OpenFeign also doesn't support reactive clients, they have created HTTP interface.

The HTTP interface has only been around since Spring Framework 6, so it's very young. Here, the team must decide whether to use the proven OpenFeign or the new HTTP interface. For this reason, the use of both libraries will now be demonstrated using one example each.

OpenFeign under Spring Boot

If you want to use OpenFeign under Spring Boot, a dependency to Spring Cloud OpenFeign is required. Then an interface has to be declared, which can look like this:

```
@FeignClient( name = "profiles", url = "http://localhost:8080" )
public interface Profiles {
  @GetMapping( "/api/profiles" )
  List<Profile> findAll();

  @GetMapping( "/api/profiles/{id}" )
  Optional<Profile> findById( @PathVariable long id );
}
```

The `@FeignClient` annotation specifies that a proxy object should be generated for this interface. The `name` attribute allows the data type to be named and further configured later. The `url` attribute can be used to set the URL for simple cases, but in production systems, this is typically done differently: the attribute remains unassigned, and the URL is determined at runtime via *Service Discovery*, an automatic service detection mechanism in the network.

Next, methods are declared and annotated within the interface. The method names don't need to be semantic. Reactive data types aren't allowed as return types. The annotations used are the same as in Spring Web MVC. This can be beneficial, but it can also be problematic because not all annotations that are allowed and useful on the server side are suitable for use on the client side.

To use the interface, you can simply inject it, as shown in the following demo program:

```
@SpringBootApplication
@EnableFeignClients
public class RestClientApplication {
  RestClientApplication( Profiles profiles ) {
    System.out.println( profiles.findAll() );
  }
  public static void main( String[] args ) {
    SpringApplication.run( RestClientApplication.class, args );
  }
}
```

The annotation `@EnableFeignClients` leads to proxy generation so that the constructor really gets an implementation of the interface.

HTTP Interface

The new Spring library HTTP interface can also be used to generate an interface, but two differences stand out:

```
@HttpExchange( url = "/api/profiles" )
public interface Profiles {
  @GetExchange
  List<Profile> findAll();

  @GetExchange( url = "/{id}" )
  Optional<Profile> findById( @PathVariable( "id" ) long id );
}
```

At the interface, the `@HttpExchange` annotation specifies the root path, and the `@GetExchange` annotation on the methods determines the HTTP method. The method names aren't semantic. The return types are known. For example, an `Optional` can be used or a list, and the data types `Mono` and `Flux` are also permitted if the reactive API is to be used.

For the interface to be implemented at runtime, the Spring HTTP interface takes a slightly different approach than OpenFeign. An annotation on the interface isn't enough; a bean of type `Profiles` must be created. This is done via a `@Configuration` and may look like this:

```
@Configuration( proxyBeanMethods = false )
public class HttpClientConfig {

  @Bean
  Profiles profileClient() {
    var client = WebClient.builder().baseUrl( "…" ).build();
    var proxyFactory = HttpServiceProxyFactory.builder(
        WebClientAdapter.forClient( client )
    ).build();
    return proxyFactory.createClient( Profiles.class );
  }
}
```

First, the `WebClient` instance is built and preconfigured, for example, with the path. An `HttpServiceProxyFactory` helps to build the `Profiles` implementation. To complete this, it's connected to the `WebClient`. It can be seen that types from Spring WebFlux are used again, so the following dependency is necessary when using it:

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-webflux</artifactId>
</dependency>
```

The dependency also starts a web server, which is of course not necessary for a pure client; a configuration property is a way to disable the server:

```
spring.main.web-application-type=NONE
```

Overall, declarative clients are a big advantage. OpenFeign is supported by many generators: We discussed the OpenAPI Generator briefly in Section 9.14.7, and *https://openapi-generator.tech/docs/generators/java* lists OpenFeign as a library. The HTTP Interface project is new; there are no client generators for it yet, but these should also follow soon.

We've learned about different possibilities for a REST client in this section. It's best to have a type definition of the JSON documents and then generate containers. If we declare a Java interface, then a runtime generated implementation can handle the communication.

You can even get a bit more abstract with *Spring Cloud Gateway* (*https://cloud.spring.io/spring-cloud-gateway/*). This project implements a kind of translator and is useful in the microservices environment. There, a server often takes a request, but forwards it directly to another node. This forwarding can be expressed declaratively, without much programming effort.

9.19 Summary

In this chapter, we've covered all the fundamental components of RESTful web services. The client sends a request with data, which is then processed by the server and returned as a response. Additionally, we explored security-related topics such as authentication. We also delved into various techniques for accessing web services from the client side.

If you want to delve deeper, you can explore the following topics:

- We haven't yet discussed *roles* that Spring can simply query declaratively in the controllers. For example, if a logged-in user has the Administrator role, certain method calls are allowed that are inaccessible to a simple user. This role check can be done declaratively in Spring. The reference documentation at *https://docs.spring.io/spring-security/reference/servlet/authorization/method-security.html* explains how this works.
- Another topic that this book almost completely omits is *HATEOAS*—and thus the question of how to link related resources. Spring offers its own project for building Hypertext Application Language (HAL) documents, *https://spring.io/projects/spring-hateoas*.
- In addition, very relevant in practice is the *reduction of bandwidth*, which causes costs, especially for cloud providers. With special headers, such as the *ETag header*, the server can assign an identifier for resources so that the client doesn't have to fetch unchanged resources that are in the cache again. The topic of caching is extensive and continues with headers such as *If-Match*, *Expires*, *Last-Modified*, and so on.
- While the early SOAP-based web services have become less popular, the *Spring Web Services* project (*https://spring.io/projects/spring-ws*) still offers excellent integration with the Spring Framework.
- As an alternative to REST, *gRPC* is another communication option that uses the HTTP/2 protocol and compact binary format for document transmission, resulting in generally faster transmission compared to JSON due to reduced overhead. While it's possible to use a binary format for document exchange in RESTful web services, the focus on resource modeling in REST differs from the operation focus of gRPC. To use a binary format, a `ProtobufHttpMessageConverter` can be included.
- Another exciting technology is *GraphQL*, which the Spring fro GraphQL project (*https://spring.io/projects/spring-graphql*) implements.
- There is an alternative to direct computer-to-computer communication in the form of messaging systems that use a broker. This approach enables excellent scaling and fail-safety. Spring provides support for this through *Spring AMQP* (*https://spring.io/projects/spring-amqp*).

Chapter 10

Logging and Monitoring

System monitoring relies heavily on logging and monitoring, as they are critical components in optimizing system performance. These tools enable us to identify and address issues swiftly, enhancing our ability to maintain optimal system functionality. In this chapter, we delve into logging once again, exploring *Spring Boot Actuator* and a time-series database.

10.1 Logging

We've already created a log with *Simple Logging Facade for Java (SLF4J)* in Chapter 1, Section 1.4.2. We did log output to see things such as where the Spring Framework calls the constructor or caching methods in the proxy. In this section, we'll look at a few more details about logging.

10.1.1 Why Create a Protocol?

Logging is an essential aspect of system monitoring as it helps developers track state changes, detect failures, and identify irregularities in a program. In production code, it's often difficult to use the debugger to identify errors, so logging provides a way to track program behavior and identify issues more easily. For instance, when a program generates exceptions, it's essential to report the error line along with contextual information. Stack traces can be helpful in identifying the class and line that caused the error, and it's crucial to determine whether to include them in the log output.

In microservice architectures, logs are particularly helpful in identifying communication failures between services. For example, when computers communicate with each other, a timeout could occur, resulting in a failure. This would be visible in the log stream, which can be monitored and analyzed to identify and address the issue.

In addition to identifying failures and errors, log messages can also be used to detect irregularities in the program. For example, if a server receives many requests, it could be an indication of an attack. HTTP requests from web servers can be logged to trace the IP addresses and resources that were requested, which can serve as an audit.

Finally, log messages can also contain profiling information that tracks the start and end of program chunks. By reading this information, developers can determine how much time a program piece has needed, which can help optimize system performance.

Overall, there are many reasons to include logging in a program, and it's an essential tool for system monitoring and optimization.

10.1.2 Log Group

As we've previously discussed, we've used the SLF4J facade to write log outputs. During this process, we've also learned about log levels. To refresh our memory, a *log level* indicates the importance or urgency of a log output to be written to the log stream. We can set log levels for packages and types by using the `logging.level` prefix.

Spring Boot can group different packages into *log groups*, for example, like this:

```
logging.group.web=org.springframework.http, org.springframework.web
```

New log groups start with prefix `logging.group`. Next comes the name of the group, followed by a comma-separated list of packages, parent packages, or specific types. Later, the log level for the entire group can be set at once:

```
logging.level.web=trace
```

In this case, the log level `trace` is passed to the two group elements.

Spring Boot itself defines two groups:

- **`web`**
  `org.springframework.core.codec`, `org.springframework.http`, `org.springframework.web`, `org.springframework.boot.actuate.endpoint.web`, `org.springframework.boot.web.servlet.ServletContextInitializerBeans`
- **`sql`**
  `org.springframework.jdbc.core`, `org.hibernate.SQL`, `org.jooq.tools.LoggerListener`

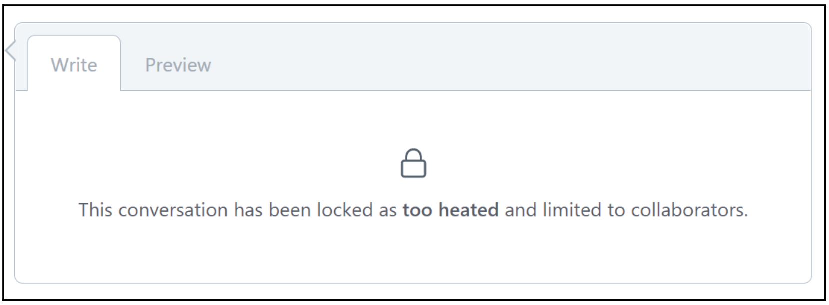
10.2 Logging Implementation

We work in Spring Boot with SLF4J, but not with the application programming interface (API) of the actual logging provider. We usually don't notice which logging implementation is doing the actual work—that is completely abstracted by Spring Boot.

The Spring team has chosen *Logback* (*https://logback.qos.ch*) as an implementation. Logback is the successor of the popular *Log4j 1.x* library. Logback was followed by *Log4j 2* (*https://logging.apache.org/log4j/2.x*), but the Spring team hasn't yet moved to Log4j 2, even though the topic keeps coming up as Figure 10.1 proves.[1]

---

[1] *https://github.com/spring-projects/spring-boot/issues/16864*



This conversation has been locked as **too heated** and limited to collaborators.

Figure 10.1 Heated Discussion about Switching from Logback to Log4j 2

10.2.1 Conversion to Log4j 2

Although Spring Boot uses Logback as a logger provider by default, Log4j 2 can easily be brought in. This is done by first taking out `spring-boot-starter-logging` and then including `spring-boot-starter-log4j2`:

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-****</artifactId>
  <!-- remove logback -->
  <exclusions>
    <exclusion>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-logging</artifactId>
    </exclusion>
   </exclusions>
</dependency>

<!-- add log4j2 -->
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-log4j2</artifactId>
</dependency>
```

10.2.2 Logging Pattern Layout

Spring configures the loggers with a pattern layout. An example output looks like this:

```
2024-10-27T18:43:29.148+02:00  INFO 22280 --- [ main] ↩
c.t.s.SpringShellApplication: No active profile set.
```

The default logging format of Spring Boot follows a specific pattern. It starts with a timestamp, followed by the log level, which, in this case, is `INFO`. The process ID is then displayed, followed by the thread name, which, in our case, is `main`. Next, the logger name is written, which is usually the same as the class name. Finally, we have the actual log message and possibly a stack trace.

This pattern is configurable and defined for Logback in the *defaults.xml* file. There are two patterns defined: one for console output and a separate pattern for log files. For console output, the pattern looks like this:

```
<property name="CONSOLE_LOG_PATTERN" value="${CONSOLE_LOG_PATTERN:-%clr(%d{${LOG_
DATEFORMAT_PATTERN:-yyyy-MM-dd'T'HH:mm:ss.SSSXXX}}){faint} %clr(${LOG_LEVEL_
PATTERN:-%5p}) %clr(${PID:- }){magenta} %clr(---){faint} %clr([
%15.15t]){faint} %clr(%-40.40logger{39}){cyan} %clr(:){faint} %m%n${LOG_
EXCEPTION_CONVERSION_WORD:-%wEx}}"/>
```

Let's go through the log output with the format again in detail. The pattern in the XML file can be decomposed into the components that are shown in Table 10.1.

Log Fragment	Meaning
`%clr(%d{${LOG_DATEFORMAT_PATTERN:-yyyy-MM-dd'T'HH:mm:ss.SSSXXX}}){faint}`↩	`%clr`: Color. With `$`, there is a fallback to the variable that defines the pattern. If this isn't defined, there is a default after the colon. `{faint}` is a faded color that isn't always visible.
`%clr(${LOG_LEVEL_PATTERN:-%5p})`↩	`-%5p`: Log level right-justified.
`%clr(${PID:- }){magenta}` ↩	`PID:-`: PID or space.
`%clr(---){faint}` ↩	Three strokes.
`%clr([%15.15t]){faint}` ↩	`t`: Thread.
`%clr(%-40.40logger{39}){cyan}` ↩	`logger`: Name of the logger, class name.
`%clr(:){faint}` ↩	Faded colon.
`%m%n${LOG_EXCEPTION_CONVERSION_WORD:-%wEx}}`	`%m`: Log message. `%n`: Line break. `%wEx`: Stack trace, if available.

Table 10.1 Default Logger Configuration for Logback

For Log4j 2, the settings look similar.[2]

---

[2] *https://github.com/spring-projects/spring-boot/blob/main/spring-boot-project/spring-boot/src/main/resources/org/springframework/boot/logging/log4j2/log4j2.xml*

10.2.3 Change the Logging Configuration

Spring Boot defines a whole set of generic configuration properties that are mapped to the corresponding logger implementations. The log settings are located under the `logging` root element:

- `logging.file.name`
  Logging to a given file instead of the console.
- `logging.file.path`
  Path for the log file (*spring.log* by default).
- `logging.pattern.console`
  Pattern for log messages on the console.
- `logging.pattern.file`
  Pattern for log messages in the log file.

There are quite a few other things you can configure. The Spring Boot reference documentation elaborates on this.[3]

10.2.4 Banner

By default, after starting up, Spring Boot displays a banner. However, we disabled the banner early on by setting a configuration property. The text for the banner is built-in and comes from a Java archive, but it can be easily replaced with custom text. To define your own banner, simply create a text file named *banner.txt*, and place it directly in the source class path under *src/main/resources*.

Tip

There are various websites that compose text from letters, numbers, and special characters into large decorative letters. A well-known program is *FIGlet* (*www.figlet.org*). There are command-line tools and various web pages that generate a large text from a text and a chosen FIGlet font—there are dozens of them. Here's an example of the text "Spring":[4]

```
o-o
|              o
 o-o  o-o  o-o   o-o  o--o
    | |  | |   | |  | |  |
o--o  O-o  o   | o  o o--O
      |                  |
      o               o--o
```

The lines can be copied directly into a separate file *banner.txt*.

---

[3] *https://docs.spring.io/spring-boot/docs/current/reference/html/spring-boot-features.html#boot-features-custom-log-configuration, https://docs.spring.io/spring-boot/docs/current/reference/html/howto-logging.html, https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#features.logging*

[4] *https://patorjk.com/software/taag/#p=display&f=Tinker-Toy&t=Spring*

Variables in Banner

In the *banner.txt* file, expressions with `${…}` can access configuration properties. This can be used to display certain information directly at startup, such as the title of the application (`${application.title}`) or even the Spring Boot version (`${spring-boot.version}`). In addition, ANSI colors can be set, such as `${Ansi.GREEN}`. (For an overview, see *http://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-banner*.)

Spring Boot automatically detects if the console recognizes colors and sets `spring.output.ansi.enabled=DETECT`. This means that if the console doesn't support colors, ANSI control codes for colors won't be displayed. Possible assignments are `ALWAYS`, `DETECT`, and `NEVER`. This is one of the important differences between logging to files and logging to the console. When outputs go to files, colors are out of place and also not set.

We've already seen in Chapter 1, Section 1.4.1, that this is done by `spring.main.banner-mode=off`. An alternative is to control the mode via `SpringApplication` and the `setBannerMode(Banner.Mode)` method.

10.2.5 Logging at Start Time

When an application starts, Spring Boot logs additional output very early, for example, about the set profile. These startup outputs can be disabled. If this isn't desired, you can set the setting via a configuration property, for example, in *application.properties*:

```
spring.main.log-startup-info=false
```

Alternatively, the property can be set via code:

```
var app = new SpringApplication( Date4uApplication.class );
app.setLogStartupInfo( false );
app.setBannerMode( Banner.Mode.OFF );
app.run( args );
```

10.2.6 Testing Written Log Message

Logging can be tested, and Spring Boot provides a way to do this. We can test whether an application has written something to the log stream. The following example demonstrates this: the `NiceGuy` class has a method that writes an `info` message to the log stream, and the `LogTest` test class accesses this method:

```
import org.assertj.core.api.Assertions;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import org.slf4j.LoggerFactory;
import org.springframework.boot.test.system.CapturedOutput;
import org.springframework.boot.test.system.OutputCaptureExtension;
```

```
class NiceGuy {
  void doWhat?() {
    LoggerFactory.getLogger( getClass() ).info( "nice guys finish last" );
  }
}

@ExtendWith( OutputCaptureExtension.class )
public class LogTest {

  private NiceGuy niceGuy = new NiceGuy();

  @Test void testLog( CapturedOutput output ) {
    niceGuy.doWhat?();
    Assertions.assertThat( output.getOut() ).contains( "finish last" );
  }
}
```

No slice test or annotation `@SpringBootTest` is needed for this test, just JUnit extension `@ExtendWith(OutputCaptureExtension.class)`. Then, when we instantiate `NiceGuy`—usually, we would inject the component—we can call the `niceGuy.doWhat?` method, which should result in log output. This can be tested. A call to `output.getOut()` will return the log messages, and AssertJ can use `contains(…)` to provide an answer as to whether `"finish last"` is included.

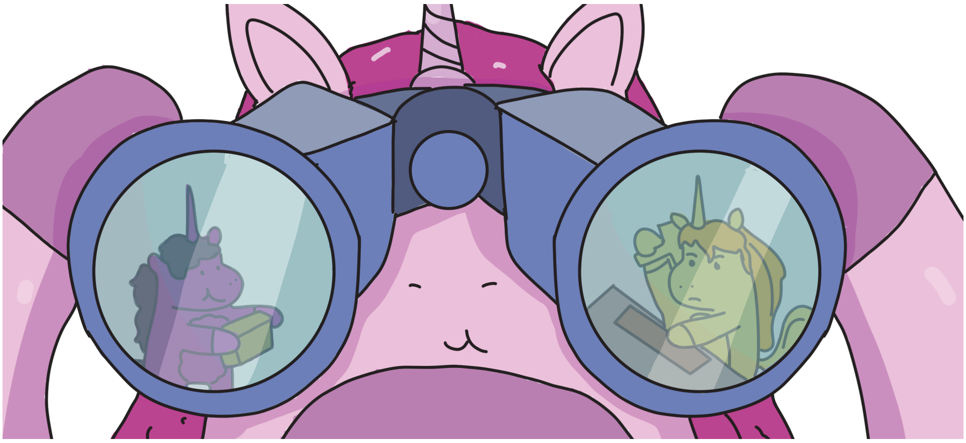
10.3 Monitor Applications with Spring Boot Actuator

Server applications are crucial, and they need to run efficiently without any issues. However, problems can always occur, so it's essential to monitor the health of the application. Admins require information about the application's status, such as memory usage, number of aborted transactions, and other important metrics.

To monitor Spring Boot applications from outside, a separate project called *Spring Boot Actuator* exists. The Actuator provides several endpoints that expose useful information about the running application. The reference documentation at *https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#actuator* provides more details about the Actuator, including how to configure it and how to access its endpoints.

Tip

If you want to see which services are currently unavailable, you can take a look at *https://downdetector.com*.



10.3.1 Determine the Health Status via the Actuator

Even if most of us don't build applications like Twitter or Netflix, it's still important to know if your own application has parameters that are out of the ordinary. That's why Spring Boot Actuator provides *Actuator endpoints*, which we can use to get specific information from the applications, such as memory usage.

Actuator endpoints can provide this data via HTTP (this is the normal case) or via *Java Management Extensions* (JMX). Today, JMX isn't that important anymore, so we'll ignore it in the following.

Set a Dependency on Actuator

The Actuator is a product of the Spring Boot family and doesn't come from the Spring Framework. To be able to use the Actuator, we put a simple dependency in the project object model (POM) file:

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

Note

Because we're using the Actuator with HTTP, there must be a dependency on `org.springframework.boot:spring-boot-starter-web`. This should be the default.

http://localhost:8080/actuator

If you start the Spring Boot program with the new dependency, *http://localhost:8080/actuator* returns a JavaScript Object Notation (JSON) document. This shows all the current endpoints:

```
{
  "_links":{
    "self":{
      "href":"http://localhost:8080/actuator",
      "templated":false
    },
    "health":{
      "href":"http://localhost:8080/actuator/health",
      "templated":false
    },
    "health-path":{
      "href":"http://localhost:8080/actuator/health/{*path}",
      "templated":true
    }
  }
}
```

Actually, there isn't much to it. There is the Actuator itself, as self-reference, and the endpoint `health`. We can query some more information, however. Following the link at *http://localhost:8080/actuator/health*, the result is just a `{"status": "UP"}`.

Although there are numerous Actuator endpoints available, most of them are deactivated. This is because it would pose a significant security issue if a Spring Boot application were to freely provide all of its information to the outside world without being requested to do so.

Configure the HTTP Actuator

The HTTP Actuator can be configured extensively. For example, you can specify which Actuator endpoints should be enabled at all. You can also specify the root path, which is `/actuator` by default. The configuration property `management.endpoints.web.base-path=/NEW-PATH` redefines the path.

Of course, the server port can be changed or Transport Layer Security (TLS) encryption can be set. More details can be found in the reference documentation of the project at *https://docs.spring.io/spring-boot/docs/current/reference/html/production-ready-monitoring.html*.

10.3.2 Activation of the Endpoints

For an Actuator endpoint to publish data, the data must be collected—Spring Boot Actuator calls this *enabled*. If an endpoint isn't enabled, the data can't be released to the outside world.

By default, all Actuator endpoints that Spring Boot defines are enabled. (There is one small exception, and that is an Actuator endpoint for a shutdown. You would have to enable it with `management.endpoint.shutdown.enabled=true`.)

That data is collected is one part of the story; the other is that the data is also given to the outside world, which Spring Boot Actuator calls *exposed*. If data isn't exposed, it's inaccessible from the outside. This can be controlled via a configuration property. To expose Actuator endpoints, put something like the following in *application.properties*:

```
management.endpoints.web.exposure.include=*
```

The asterisk is a wildcard that says: "Expose all." Actuator endpoints can be selectively enabled by listing them in a comma-separated format, for example, `…info=info,health`. The reference documentation lists which endpoints are named what. The IDs are as follows:

- `auditevents`
- `beans`
- `caches`
- `conditions`
- `configprops`
- `env`
- `flyway`
- `health`
- `httpexchanges`
- `info`
- `integrationgraph`
- `loggers`
- `liquibase`
- `metrics`
- `mappings`
- `quartz`
- `scheduledtasks`
- `sessions`
- `shutdown`
- `startup`
- `threaddump`
- `heapdump`
- `logfile`
- `prometheus`

Alternatively, all endpoints can be selected with `*`, and then you can use `exclude` to eliminate certain endpoints after all:

```
management.endpoints.web.exposure.exclude=env,beans
```

Normally, the better variant is to precisely enumerate the desired endpoints when `including`. This reduces security problems and also makes the application minimally faster because if less data is collected, there is less load.

In the following, we assume that we've unlocked everything with the asterisk, that is, published it.

10.3.3 Info Supplements

The `info` endpoint path can be used to pass any proprietary information from the application to the outside. For this purpose, data is stored following the `info` prefix, such as the following two entries, which can be located in *application.properties*:

```
info.build.author=Crazy Joe
info.build.version=12.23454
```

This lists the build information under the `info` endpoint. It's built as a JSON document. Certain build information can be automatically read from the Maven or Gradle file in advance, for example, using the Spring Boot Maven plugin. It's configured as follows:

```
<plugin>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-maven-plugin</artifactId>
  <executions><execution>
    <id>build-info</id>
    <goals><goal>build-info</goal></goals>
  </execution></executions>
</plugin>
```

The plugin creates a *META-INF/build-info.properties* file. It's included by Spring Boot on its own. This is handy because if there is a deployment of the application in the cloud, and there is an error, then it must be known which version is running. With the plugin, the data can be written automatically.

10.3.4 Parameters and JSON Returns

Many Actuator endpoints can be passed further data, and they have path variables. These path variables can be easily identified in the overall listing using the familiar syntax of the URI template:

```
"caches-cache":{
 "href":"http://localhost:8080/actuator/caches/{cache}",
 "templated":true
},
"health-path":{
 "href":"http://localhost:8080/actuator/health/{*path}",
 "templated":true
}, …
"loggers-name":{
 "href":"http://localhost:8080/actuator/loggers/{name}",
 "templated":true
},
"metrics-requiredMetricName":{
 "href":"http://localhost:8080/actuator/metrics/{requiredMetricName}",
 "templated":true
}, …
```

The information represented by `{cache}` can be used to retrieve specific information from a cache. Similarly, for loggers, `{name}` denotes that specific information on a particular logger can be queried. For further understanding, two concrete examples are available at:

- *http://localhost:8080/actuator/loggers/com.tutego*
- *http://localhost:8080/actuator/metrics/jvm.memory.max*

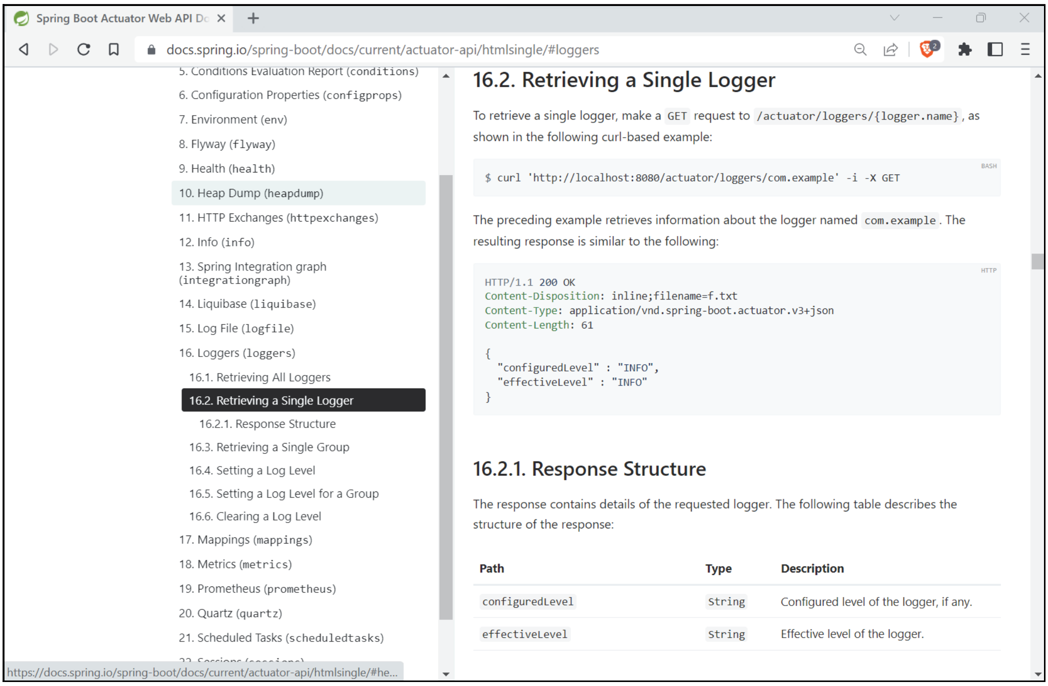
The first example queries the log level of the `com.tutego` package and the second queries a metric of the Java virtual machine (JVM), namely JVM Max Memory.

So, there isn't necessarily a collection of data behind the endpoints, but endpoints can also be parameterized via a path variable. The parameters and the exact JSON structure of the returns are explained at *https://docs.spring.io/spring-boot/docs/current/actuator-api/html/* (see Figure 10.2).

The website documents how an endpoint is called and which JSON documents come back. Loggers have the special feature that log levels can also be set. The documentation states the following:

```
$ curl 'http://localhost:8080/actuator/loggers/com.example' -i -X POST \
    -H 'Content-Type: application/json' \
    -d '{"configuredLevel":"debug"}'
```

Although the format of the response may not matter to humans, it's essential for tools. Monitoring tools or other programs that periodically call these endpoints, for instance, require knowledge of the response format.


Spring Boot Actuator Web API Do
docs.spring.io/spring-boot/docs/current/actuator-api/htmlsingle/#loggers
5. Conditions Evaluation Report (conditions)
6. Configuration Properties (configprops)
7. Environment (env)
8. Flyway (flyway)
9. Health (health)
10. Heap Dump (heapdump)
11. HTTP Exchanges (httpexchanges)
12. Info (info)
13. Spring Integration graph (integrationgraph)
14. Liquibase (liquibase)
15. Log File (logfile)
16. Loggers (loggers)
16.1. Retrieving All Loggers
16.2. Retrieving a Single Logger
16.2.1. Response Structure
16.3. Retrieving a Single Group
16.4. Setting a Log Level
16.5. Setting a Log Level for a Group
16.6. Clearing a Log Level
17. Mappings (mappings)
18. Metrics (metrics)
19. Prometheus (prometheus)
20. Quartz (quartz)
21. Scheduled Tasks (scheduledtasks)
https://docs.spring.io/spring-boot/docs/current/actuator-api/htmlsingle/#he...
BASH
HTTP

16.2. Retrieving a Single Logger

To retrieve a single logger, make a `GET` request to `/actuator/loggers/{logger.name}`, as shown in the following curl-based example:

```
$ curl 'http://localhost:8080/actuator/loggers/com.example' -i -X GET
```

The preceding example retrieves information about the logger named `com.example`. The resulting response is similar to the following:

```
HTTP/1.1 200 OK
Content-Disposition: inline;filename=f.txt
Content-Type: application/vnd.spring-boot.actuator.v3+json
Content-Length: 61

{
  "configuredLevel" : "INFO",
  "effectiveLevel" : "INFO"
}
```

16.2.1. Response Structure

The response contains details of the requested logger. The following table describes the structure of the response:

Path	Type	Description
`configuredLevel`	`String`	Configured level of the logger, if any.
`effectiveLevel`	`String`	Effective level of the logger.

Figure 10.2 Actuator Endpoint Responses

10.3.5 New Actuator Endpoints

Many Actuator endpoints are predefined, but it's easy to program and add your own Actuator endpoints. To accomplish this, a new Spring-managed bean must be programmed and annotated:

```
@Component
@WebEndpoint( id="stat" )
public class ActuatorStatisticsBean {
  @ReadOperation
  public int numberOfProfiles() {
    return 1337;
  }
}
```

Then, various HTTP methods can be used to retrieve information or also to modify or delete it. Further details are given in the reference documentation.[5]

---

[5] *https://docs.spring.io/spring-boot/docs/current/reference/html/actuator.html#actuator.endpoints.implementing-custom*

10.3.6 Am I Healthy?

At *http://localhost:8080/actuator/health*, you can find information about the “health state” of the application. By default, there is only one status there, and it’s a small JSON object:

```
{"status": "UP"}
```

The status can be `UP` or `DOWN`.

While it may seem like a joke that your application reports whether it’s up or down, this approach can be more useful for subcomponents. For instance, when Spring Boot reports the status of a database, it provides more valuable information about the health of the application.

managment.endpoint.health.show-components

The amount of information can be increased with configuration property `management.endpoint.health.show-components=always`. The assignment with `always` shows a somewhat more extensive JSON document:

```
{
  "status":"UP",
  "components":{
    "diskSpace":{
      "status":"UP"
    },
    "ping":{
      "status":"UP"
    }
  }
}
```

In addition to the `UP` status information, there is a subelement named `components`. This component contains the disk space status and information about the ping.

management.endpoint.health.show-details

For security reasons, only the `UP` and `DOWN` remain. But with configuration property `management.endpoint.health.show-details=always`, further information can be displayed. The result is as follows:

```
{ "status":"UP", "components":{
 "diskSpace":{
  "status":"UP",
  "details":{
    "total":510770802688, "free":324784771072,
    "threshold":10485760, "exists":true
  }
}, "ping":{ "status":"UP" } } }
```

In addition to the status, you can see further details, such as `total`, `free`, `threshold`, and `exists`—this is much more than just the information `UP` or `DOWN`.

Valid values for `show-details` and `show-components` are `never` (default), `always`, and `when-authorized`. The value `never` is the default, so we didn't see anything at the beginning. Specifying `when-authorized` might be the most reasonable in practice. It ensures that data is only displayed when a user is properly authenticated. Actuator endpoints run regularly through *Spring Security* (more details at *https://docs.spring.io/spring-boot/docs/current/reference/html/actuator.html#actuator.endpoints.health*).

10.3.7 HealthIndicator

Spring Boot implements the information about `UP` and `DOWN` with all the respective details using a `HealthIndicator`.[6] This is an interface with a single `Health health()` method, and it returns `Health.up()` or `Health.down()`, depending on the health state.

Various `HealthIndicator` implementations for infrastructure services are added via the auto-configurations:

- `DiskSpaceHealthIndicator`
- `DataSourceHealthIndicator`
- `MongoHealthIndicator`
- `ElasticsearchRestHealthIndicator`

Note

There are other implementations as well, so see *https://docs.spring.io/spring-boot/docs/current/reference/html/actuator.html#actuator.endpoints.health.auto-configured-health-indicators* for a full list.

The implementation isn't difficult, and everyone is invited to study the source code of `DiskSpaceHealthIndicator`, for example.[7]

10.3.8 Metrics

Actuator endpoints in Spring Boot expose various types of data from within the application. However, not all of this data is useful for monitoring purposes. For instance, in the development phase, information about Spring-managed beans may be interesting, but it doesn't necessarily indicate anything about the performance or stability of an application. To gather meaningful insights into an application's performance, it's necessary to collect *metrics*. Metrics involve tracking certain values at specific points in time, such as memory usage, thread count, HTTP requests processed, and garbage collection failures. These metrics are important for monitoring an application's health and can help identify potential issues.

---

[6] *https://docs.spring.io/spring-boot/docs/current/api/org/springframework/boot/actuate/health/HealthIndicator.html*

[7] *https://github.com/spring-projects/spring-boot/blob/main/spring-boot-project/spring-boot-actuator/src/main/java/org/springframework/boot/actuate/system/DiskSpaceHealthIndicator.java*

/actuator/metrics

The Actuator endpoint *http://localhost:8080/actuator/metrics* only gives us a list of keys, but no data itself:

```
{
  "names":[
    "application.ready.time",
    "application.started.time",
    "disk.free",
    "disk.total",
     …
    "executor.queued",
    "http.server.requests",
    "jvm.buffer.count",
     …
    "tomcat.sessions.rejected"
  ]
}
```

The real information is retrieved via the key, which is used as a path variable. For example, *http://localhost:8080/actuator/metrics/jvm.memory.max* specifically queries `jvm.memory.max`. An explanation of the individual keys is again provided in the reference documentation at *https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#actuator.metrics.supported*.

10.4 Micrometer and Prometheus

It's essential to carefully consider which data to publish. On one hand, collecting data that isn't analyzed can result in unnecessary performance overhead. On the other hand, including unimportant data can quickly distract from identifying and solving the core problem. This section will explore the following areas:

- Technologies that can be used for data collection
- How to publish and store the collected data in the long term
- Methods for visualizing the collected data

10.4.1 Micrometer

Spring Boot relies on an open-source library called *Micrometer* (*https://micrometer.io*) to collect, describe, and provide metrics via its API. Users of the Micrometer library can be identified by importing types from the `io.micrometer` package.

We can examine a snippet of the source code of the `JvmMemoryMetrics` class for capturing JVM memory metrics. The code comes from Micrometer and not from Spring Boot. That is, Micrometer can already capture this data itself:

```
@NonNullApi
@NonNullFields
public class JvmMemoryMetrics implements MeterBinder {
  …
  @Override
  public void bindTo( MeterRegistry registry ) {
    for ( MemoryPoolMXBean memoryPoolBean :
              ManagementFactory.getPlatformMXBeans(
                                   MemoryPoolMXBean.class ) ) {
      String area = MemoryType.HEAP.equals(
          memoryPoolBean.getType() ) ? "heap" : "nonheap";
      Iterable<Tag> tagsWithId = Tags.concat( tags, "id",
                           memoryPoolBean.getName(), "area", area );
      …
      Gauge.builder( "jvm.memory.max", memoryPoolBean,
        ( mem ) -> getUsageValue( mem, MemoryUsage::getMax ) )
          .tags( tagsWithId )
          .description( "The maximum amount of memory in bytes "+
                        "that can be used for memory management" )
          .baseUnit( BaseUnits.BYTES ).register( registry );
    }
  }
}
```

Listing 10.1 https://github.com/micrometer-metrics/micrometer/blob/main/micrometer-core/src/main/java/io/micrometer/core/instrument/binder/jvm/JvmMemoryMetrics.java

Method `bindTo(…)` is used to bind information to a registry. Subsequently, at defined intervals, the `MeterRegistry` will retrieve the current state of the information from all registered sources. The memory usage data is obtained from an MXBean, which is a part of Java SE. Texts and descriptions are then added to the information source, making it ready for use.

Spring Boot and Micrometer

We can also provide such *gauges* ourselves—and this is something that Spring Boot then does. Let's look at a snippet from the Spring Boot class `StartupTimeMetricsListener`:

```
public class StartupTimeMetricsListener
    implements SmartApplicationListener {
  …
  private final MeterRegistry meterRegistry;
  …
  public StartupTimeMetricsListener(
      MeterRegistry meterRegistry, String startedTimeMetricName,
      String readyTimeMetricName, Iterable<Tag> tags ) {
    this.meterRegistry = meterRegistry;
    this.startedTimeMetricName = startedTimeMetricName;
    this.readyTimeMetricName = readyTimeMetricName;
    this.tags = Tags.of( tags );
  }

…

  private void onApplicationStarted( ApplicationStartedEvent event ) {
    registerGauge( this.startedTimeMetricName,
  "Time taken (ms) to start the application", event.getTimeTaken(),
                   event.getSpringApplication() );
  }

  private void registerGauge( String name, String description,
                              Duration timeTaken,
                              SpringApplication springApplication ) {
    if ( timeTaken != null ) {
      Iterable<Tag> tags = createTagsFrom( springApplication );
      TimeGauge.builder( name, timeTaken::toMillis,
          TimeUnit.MILLISECONDS )
               .tags( tags )
               .description( description )
               .register( this.meterRegistry );
    }
  }
}
```

Listing 10.2 https://github.com/spring-projects/spring-boot/blob/main/spring-boot-project/spring-boot-actuator/src/main/java/org/springframework/boot/actuate/metrics/startup/StartupTimeMetricsListener.java

Further examples are provided at *https://github.com/spring-projects/spring-boot/tree/main/spring-boot-project/spring-boot-actuator/src/main/java/org/springframework/boot/actuate/metrics*.

Sprint Boot’s Actuator project accesses and can display Micrometer’s data.

Micrometer Publication

Collecting and storing data inside an application is just the first step toward analyzing it. The collected data needs to be made available to external monitoring solutions for further processing and analysis. To accomplish this, Micrometer and Spring Boot provide add-on modules that can be used to make the collected data available to external monitoring solutions.

These external solutions pick up the data, store it for later analysis, and provide various tools and features for visualizing and analyzing the data. There are many products for which Spring can publish the data:

- AppOptics
- Atlas
- Datadog
- Dynatrace
- Elastic
- Ganglia
- Graphite
- Humio
- Influx
- JMX
- KairosDB
- New Relic
- OpenTelemetry (default)
- Prometheus*
- SignalFx
- Simple
- Stackdriver
- StatsD
- Wavefront

Again, details are provided in the reference documentation.[8] In the following, we’ll take a closer look at Prometheus.

---

[8] *https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#actuator.metrics.export*

10.4.2 Prometheus: The Software

Prometheus (*https://prometheus.io*) is an open-source monitoring software that is widely used in the industry. It follows a pull-based approach, where it periodically fetches data from the application being monitored and stores it in a *time-series database*. This type of database is optimized for recording data at specific points in time and is highly efficient. To query and aggregate the data, Prometheus provides a query language called *Prometheus Query Language* (PromQL).

It's important to note that Prometheus requires a specific format that the Spring Boot application must prepare. To provide this format, a special Actuator endpoint is needed. As there are numerous monitoring solutions, each with their own data format, an additional dependency is required to facilitate the conversion of the data to the appropriate format in the *POM.xml*:

```
<dependency>
  <groupId>io.micrometer</groupId>
  <artifactId>micrometer-registry-prometheus</artifactId>
</dependency>
```

As an Actuator endpoint, we need to expose the following:

```
management.endpoints.web.exposure.include=prometheus
```

After restarting the application, you'll recognize the special format at *http://localhost:8080/actuator/prometheus*:

```
# HELP process_start_time_seconds Start time of the process since unix epoch.
# TYPE process_start_time_seconds gauge
process_start_time_seconds 1.667078065888E9
…
# HELP jvm_memory_max_
bytes The maximum amount of memory in bytes that can be used for memory manageme
nt
# TYPE jvm_memory_max_bytes gauge
jvm_memory_max_bytes{area="heap",id="G1 Survivor Space",} -1.0
jvm_memory_max_bytes{area="heap",id="G1 Old Gen",} 4.242538496E9
jvm_memory_max_bytes{area="nonheap",id="Metaspace",} -1.0
jvm_memory_max_bytes{area="nonheap",id="CodeCache",} 5.0331648E7
jvm_memory_max_bytes{area="heap",id="G1 Eden Space",} -1.0
jvm_memory_max_bytes{area="nonheap",id="Compressed Class Space",} 1.073741824E9
…
# HELP jvm_gc_memory_promoted_bytes_
total Count of positive increases in the size of the old generation memory pool
before GC to after GC
# TYPE jvm_gc_memory_promoted_bytes_total counter
jvm_gc_memory_promoted_bytes_total 0.0
# HELP executor_pool_max_
```

```
threads The maximum allowed number of threads in the pool
# TYPE executor_pool_max_threads gauge
executor_pool_max_threads{name="applicationTaskExecutor",} 2.147483647E9
```

It's exactly this format that Prometheus can pick up and process. Because the data only has to be machine-readable, it's quite strange that it's offered as text format.

Install Prometheus

To ensure that using Prometheus doesn't remain a theory, we run the software. Then, we configure Prometheus so that the software collects and saves the data from our Spring Boot application at regular intervals.

There are two methods to launch Prometheus—either via an Open Container Initiative (OCI) container (commonly known as Docker) or by downloading and launching the product traditionally. As the latter option is straightforward, we can choose to do that.

- **Step 1**: Go to *https://prometheus.io/download*, get the ZIP archive, and unpack it. The server is available for different operating systems.

  The directory content looks like this after unpacking (Windows):

  - *consoles*
  - *console_libraries*
  - *LICENSE*
  - *NOTICE*
  - *prometheus.exe*
  - *prometheus.yml*
  - *promtool.exe*

  Here the YAML Ain't Markup Language (YAML) file *prometheus.yml* catches the eye because it configures Prometheus, among other things, with individual data providers. For this, we need to find out our IP address so that Prometheus can connect to our server.
- **Step 2**: Open the shell, run `ipconfig`, and write down your IP address, probably something like `192.168.xx.yy`.
- **Step 3**: Open the *prometheus.yml* file. Add the following code, and insert your IP address:

  ```
  - job_name: "spring-actuator"
    metrics_path: "/actuator/prometheus".
    scrape_interval: 1s
    static_configs:
      - targets: ["MY.OWN.IP.ADDRESS:8080"]
  ```

In Prometheus, the concept of *jobs* is central to its operation. In the given code snippet, a new job is listed with a specified name, data path, host address, and port. The fetch interval is set to 15 seconds by default, but the `scrape_interval` setting can be configured to lower the fetch frequency to 1 second for faster data retrieval. This means that every second, Prometheus will access the specified endpoint, fetch the data, and store it for analysis.

Prometheus can now be started:

```
$ prometheus.exe
ts=2022-10-29T22:15:21.059Z caller=main.go:499 level=info
msg=
"No time or size retention was set so using the default time retention" duration
=15d
…
ts=2022-10-29T22:15:21.105Z caller=manager.go:943 level=info
component="rule manager" msg="Starting rule manager…"
```

If you open *http://localhost:9090/targets* in a browser, you see the data sources, that is, our Spring Boot application. If you go to **Graph** in the interface or enter "http://localhost:9090/graph" directly, a search for `jvm_memory_used_bytes` returns the data, which can be viewed in tabular form or via a graph. Because Prometheus fetches a sample every second, you can see the graph slowly "grow."

Display isn't the main focus of Prometheus because Prometheus is a time-series database, not a visualization tool. If you're interested in data visualization, you should have a look at *Grafana* (*https://grafana.com*); the connection with Prometheus can be established with a few clicks.

10.5 Summary

In this chapter, we've learned how to write and format outputs to the log stream and redirect them to files. Processing these log files is an important topic, and solutions such as the *Elastic Stack* come into play here.[9] This is especially important in a microservice architecture, where local logging is less relevant, and it's essential that the nodes send the log data to a common log server where the data can be aggregated and prepared. For more details, you can visit *www.elastic.co/elastic-stack*.

Another peculiarity in a microservice architecture is that requests often go through multiple computer nodes. It makes sense to tag messages with an ID so that the context of a cross-cutting request isn't lost and can be visualized later. We call this *distributed tracing*, and solutions for this include *Jaeger* (*www.jaegertracing.io*) and *Zipkin* (*https://zipkin.io*). Both can be integrated perfectly into Spring. *Micrometer Tracing* (*https://micrometer.io/docs/tracing*) is slowly emerging as a general standard.

The new *Observation API* offers a straightforward API for reporting information, which is briefly introduced at *https://spring.io/blog/2022/10/12/observability-with-spring-boot-3*.

---

[9] The Elastic Stack used to be called the ELK Stack, an abbreviation of the project names Elasticsearch, Logstash (data processing pipeline), and Kibana (visualization tool).

Chapter 11

Build and Deployment

In the previous chapters, we took care of many technical details such as how to address a database, how to implement RESTful web services, and so on. But we don't want our program to run in the development environment later, but in a target environment such as a server in the cloud.

11.1 Package and Run Spring Boot Programs

In this chapter, we'll look at how to build the Spring Boot program and what the different deployment mechanisms are.

11.1.1 Deployment Options

Software deployment, which involves installing and configuring software, can be achieved in various ways for Spring applications, as with any Java software. Some deployment options for Spring applications include the following:

- Building a classic Java archive (JAR) and uploading it as a JAR file, which can be executed on a Java virtual machine (JVM).
- Compiling the Java application natively, generating an EXE file, for example, under Windows. This approach is a modern technique that the Spring team has worked on to improve and has become one of the key innovations of Spring Framework 6 and Spring Boot 3.
- Deploying Spring applications as web application (WAR) files on a running servlet container or Jakarta Enterprise container. This method has become less common today. Interested parties can find corresponding paragraphs in the reference documentation.[1]
- Using Spring programs in serverless environments in the cloud, for example, as an *AWS Lambda function*. The Spring team has launched the *Spring Cloud Function project* (*https://spring.io/projects/spring-cloud-function*) for this purpose.

---

[1] *https://docs.spring.io/spring-boot/docs/current/reference/html/ howto.html#howto.traditional-deployment*

- Packaging the application as a container image, which includes the operating system and Java virtual machine (JVM) in an Open Container Initiative (OCI) container. This container image can then be uploaded to the cloud for execution.

11.1.2 Launching a Spring Boot Program via Maven

If you use Maven or Gradle, you can start Spring Boot applications directly with both build tools. For this purpose, there is a special plugin goal `spring-boot:run`. This requires at least Java 17, and the environment variable `JAVA_HOME` must be set:

```
$ set JAVA_HOME=C:\my-path-to\java\jdk17
```

After installing Maven, or using the Maven wrapper, the `spring-boot:run` plugin goal can be executed as follows:

```
$ mvn spring-boot:run
```

You can also skip the test cases. This is always advantageous if the test cases have errors because otherwise the build phase terminates, and it doesn't continue:

```
$ mvn spring-boot:run -DskipTests
```

If there are multiple classes with `main` methods in the project, the one class that distinguishes the starting class of the Spring Boot application must be specified in the project object model (POM) or Gradle file. In Maven, this takes place in the `<properties>` block:

```
<start-class>com.example.Date4uApplication2</start-class>
```

In our Date4u project, we have only one `main` method.

11.1.3 Packing a Spring Boot Program into a Java Archive

Launching via Maven is one way; another is to generate a launchable JAR. For this, you trigger the `package` phase:

```
$ mvn package
```

If there is an error in the test case, and we want to ignore it, we skip the test cases:

```
$ mvn -DskipTests package
```

Maven generates two JARs in the *target* directory:

- *date4u-0.0.1-SNAPSHOT.jar.original*
- *date4u-0.0.1-SNAPSHOT.jar*

The file with the extension *original* contains our own code. And while our compressed code is rather small, the actual executable JAR file will already be around 50 MB because everything the application references as dependencies is packed into the large JAR. This then includes the parts of the Spring Framework, Tomcat, H2, and so on.

We can start the JAR in which, of course, the corresponding Java version must also fit:

```
$ java -jar target\date4u-0.0.1-SNAPSHOT.jar
```

11.1.4 spring-boot-maven-plugin

During the package phase in Maven, three things are accomplished: first, it builds its own JAR; second, it renames the original; and third, it bundles the comprehensive JAR with all its dependencies. This "repackaging" is done by a plugin that is included in our POM:

```
<?xml version="1.0" encoding="UTF-8"?>
<project …>
  …
  <build>
    <plugins>
       <plugin>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-maven-plugin</artifactId>
      </plugin>
    </plugins>
  </build>

</project>
```

The `spring-boot-maven-plugin` defines several plugin goals, including the goal `repackage` that turns our JAR into the bootable JAR.

Plugin Goals

There are other plugin goals, which are briefly explained at *https://docs.spring.io/spring-boot/docs/current/maven-plugin/reference/html/#goals* and described in the following list:

- **`spring-boot:build-image`**
  This generates an OCI container image. The term "Docker image" is often used, but it's no longer appropriate. Since Spring Boot 3.1, there is also `spring-boot:build-image-no-fork`.
- **`spring-boot:build-info`**
  This takes selected information (e.g., the version number) from the POM file and puts it into a special properties file that the actuator endpoint publishes. We discussed this in Chapter 10, Section 10.3.3.

- **`spring-boot:help`**
  This is self-explanatory.
- **`spring-boot:repackage`**
  This generates a startable JAR.
- **`spring-boot:run`**
  This builds the software and starts it. `spring-boot:test-run` is new in Spring Boot 3.1.
- **`spring-boot:start`**
  This starts the program, just not blocking.
- **`spring-boot:stop`**
  This terminates the program started with `start`.
- **`spring-boot:process-aot`**, **`spring-boot:process-test-aot`**
  These two goals were added in Spring Boot 3.0 and are used for native compilation.

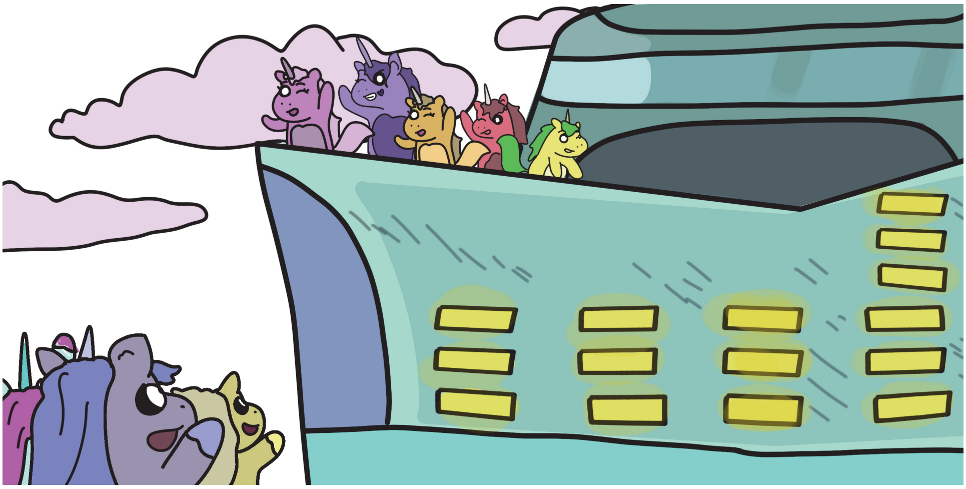
11.2 Spring Applications in the OCI Container

There are several ways to run Spring applications. One approach is to compile the application into bytecode and provide a JVM on the executing side. However, a more modern approach is to package the application as a container image using the *Open Container Initiative (OCI)* format, as we've mentioned earlier. This involves bundling the operating system, the JVM, and the Java program into a single unit. In this section, we look at how this works.

11.2.1 Container

The deployment of Java software can be challenging due to the dependencies on specific Java versions and other external factors such as the file system and database configuration. Even thanks to tools such as Maven, there may still be unforeseen conflicts when running the software on a different environment. The solution to this problem is *containerization*, which involves bundling all the dependencies of a piece of software into a container, an isolated environment with its own file system and network settings. This eliminates conflicts with other components and ensures that the software can run consistently across different environments.

Containers are an alternative to *virtualization*, which replicates an entire computer, including all hardware and software, resulting in a heavy and resource-intensive solution. Containers, on the other hand, are an abstraction layer of the operating system, allowing for lightweight deployment and efficient use of resources. While virtualization provides higher levels of isolation and security, containers offer a balance of isolation and performance, making them a popular choice for modern software deployment.



Terms: Image and Container

Applications, along with their dependencies, can be packaged into an *image*, which is essentially a template that can be used to create a container. Containers are isolated processes that run independently of the host operating system, except for a few interfaces to the underlying hardware and software. Building these images is facilitated by specific software, and running them requires different tools.

Once created, images are typically uploaded to a hub from which they can be downloaded and run on different machines. This is particularly useful in a distributed microservice architecture, where a Spring application can be packaged into an image and then deployed to various machines in the cloud. Sophisticated software can also be used to manage containers, including monitoring and automatically restarting them if they fail.

Open Container Initiative

One initiative that creates standards around containers and images is the OCI. At the heart of the OCI are the *Runtime Specification* and the *Image Specification*. This is similar to the Java platform: the Java compiler generates bytecode, and the JVM executes it. The specification of the bytecode is figuratively the Image Specification, and the JVM is the Runtime Specification.

There is a wide range of products around these two specifications. You'll often stumble across the following three products in practice:

- *Docker* (*www.docker.com*)
- *Podman* (*https://podman.io*)
- *Rancher* (*www.rancher.com*)

Docker is the first product to bring container virtualization to the mainstream. The company behind *Docker*, *Docker Inc.*, founded the OCI.

We want to stick with Docker for the examples because the product is widely used, and there is a lot of documentation and many examples. Ranger is an excellent alternative with the same command-line switches, and Podman is very popular, especially in the Linux environment.

11.2.2 Install and Use Docker

Installation packages for Docker can be found for Windows, macOS, and Linux on Docker's homepage. Three steps are required (for Windows):

1. Install *Windows Subsystem for Linux* (WSL) from *https://learn.microsoft.com/en-us/windows/wsl/install-manual#step-4---download-the-linux-kernel-update-package* (no reboot necessary).
2. Obtain an installer for Docker at *https://docs.docker.com/docker-for-windows/install*, and then install it.
3. Restart your computer.

The *Docker Desktop* (a graphical interface for Docker) and a console application are installed. After installation, you can test if there are any errors when executing the following command:

```
$ docker version
Client:
 Cloud integration: v1.0.29
 Version:           20.10.20
 …

Server: Docker Desktop 4.13.0 (89412)
 Engine:
  Version:          20.10.20
  API version:      1.41 (minimum version 1.12)
  …
 containerd:
  Version:          1.6.8
  GitCommit:        9cd3357b7fd7218e4aec3eae239db1f68a5a6ec6
 runc:
  Version:          1.1.4
  GitCommit:        v1.1.4-0-g5fd4c4d
 docker-init:
  Version:          0.19.0
  GitCommit:        de40ad0
```

OCI Images and Registries

To start a Docker container, an image must first be available. There are two ways to get an image: you can either download an existing image or build your own image using a *Dockerfile*. However, building an image from scratch can be a time-consuming process. Luckily, there are more convenient mechanisms to build an OCI image from a Spring application.

Docker Hub is the default registry for Docker, and it contains a wide variety of images for different servers and software systems. To check which images are installed, you can view them using Docker Desktop or via the command line:

```
$ docker images
REPOSITORY TAG IMAGE ID CREATED SIZE
```

For those who have never used Docker before, the list is empty.

An image can be loaded from Docker Hub with `docker pull [IMAGE_NAME]`. This doesn't happen that often because often you run `docker run [IMAGE_NAME]`, which also pulls the image from the registry, but starts right away.

The Whale Says . . .

As an example, let's obtain an image for an ASCII graphic and run it in a container:

```
$ docker run docker/whalesay cowsay tutego
Unable to find image 'docker/whalesay:latest' locally
latest: Pulling from docker/whalesay
Image docker.io/docker/whalesay:latest uses outdated schema1 manifest format. ↩
Please upgrade to a schema2 image for better future compatibility. ↩
More information at ↩
https://docs.docker.com/registry/spec/deprecated-schema-v1/
e190868d63f8: Pull complete
909cd34c6fd7: Pull complete
0b9bfabab7c1: Pull complete
a3ed95caeb02: Pull complete
00bf65475aba: Pull complete
c57b6bcc83e3: Pull complete
8978f6879e2f: Pull complete
8eed3712d2cf: Pull complete
Digest: ↩
sha256:178598e51a26abbc958b8a2e48825c90bc22e641de3d31e18aaf55f3258ba93b
Status: Downloaded newer image for docker/whalesay:latest

________
< tutego >
 --------
    \
     \
      \
                    ##        .
              ## ## ##       ==
           ## ## ## ##      ===
       /""""""""""""""""___/ ===
  ~~~ {~~ ~~~~ ~~~ ~~~~ ~~ ~ /  ===- ~~~
       \______ o          __/
        \    \        __/
          \____\______/
          ----
```

If you run `docker images` again, the loaded image appears:

```
$ docker images
REPOSITORY        TAG       IMAGE ID       CREATED       SIZE
docker/whalesay   latest    6b362a9f73eb   7 years ago   247MB
```

You can also see the images in Docker Desktop.

Docker Hub

When you call `docker run docker/whalesay`, it performs two actions. First, it downloads the image for `whalesay` from Docker Hub (*https://hub.docker.com*), which hosts more than 100,000 images that can be downloaded and run locally. These images are often preconfigured, such as servers with demo data, which makes it easy to test and try out new things. Second, the downloaded image is executed. It's worth noting that the H2 database can also be started via an OCI container, but it may not be worth it if you already have a JVM installed locally. However, if you don't have a JVM installed or require a specific version, you can easily start H2 with an OCI container.

11.2.3 H2 Start, Stop, Port Forwarding, and Data Volumes

There are several images for H2, so let's have a look at *https://hub.docker.com/r/thomseno/h2/*. The source code of the Dockerfile can be found at *https://github.com/thomseno/docker-h2*. With `docker run`, the image can be loaded and started from Docker Hub as shown:

```
$ docker run thomseno/h2
```

After startup, you can use `docker ps` (`ps` stands for *process status*) to list all started containers. This may look like this:

```
$ docker ps -a
CONTAINER ID   IMAGE         COMMAND     …

787774d3a3fd   thomseno/h2   …           …
```

In addition to `docker run`, there is a `docker stop` command that can be used to stop a container. The general call is as follows:

```
docker stop [OPTIONS] CONTAINER [CONTAINER…]
```

We can pass the container ID that comes from `docker ps`.

To stop the H2 container, we need to know its container ID:

```
$ docker ps -a
CONTAINER ID   IMAGE         COMMAND     …
787774d3a3fd   thomseno/h2   …           …

$ docker stop 39fb14e04f6b
39fb14e04f6b
```

Stopping in Docker Desktop is more convenient.

Normally, a port inside the container can't be used from outside. To access the container externally anyway, a container can be created with a *port mapping*. A port mapping is set with the `-p` option (or in the long form `--expose`). The general syntax is

```
docker run -p [HOST_PORT]:[CONTAINER_PORT] [IMAGE_NAME]
```

All port mappings of a container can also be displayed later. The format is

```
docker port [CONTAINER]
```

Let's start H2 again, and release two ports:

```
$ docker run -p 9092:9092 -p 8082:8082 thomseno/h2
```

On port `9092`, the database server receives the request; on port `8082`, the H2 console, the web interface, runs. We leave the ports unchanged.

Note

Of course, the ports must also be available on the host. If you try this with H2, you have to stop the local H2 database. In principle, however, the Spring program could already fall back on H2 in the container, and the web page at *http://localhost:8082/* also shows the H2 console.

The `docker run` statement starts the container and blocks. If you want to run the container in the background, use the `-d` option or, in the long version, `--detach`. Here's an example:

```
$ docker run -d -p 9092:9092 -p 8082:8082 thomseno/h2
d487c2d08aa84a66de6de2c0b7effcd1f4988576897259ab74d6fc2a1504a784
```

The container ID is output.

A repeated `docker ps` shows the running H2 database:

```
$ docker ps -a
CONTAINER ID   IMAGE          COMMAND     ... PORTS
          ...

787774d3a3fd   thomseno/h2   ...          ... 0.0.0.0:8082->8082/
tcp, 0.0.0.0:9092->9092/tcp ...
```

All containers, even those that have already been terminated, are displayed by `docker ps -a`:

```
$ docker ps -a
CONTAINER ID   IMAGE              COMMAND                   ...
787774d3a3fd   thomseno/h2        "/bin/sh -c 'java ${…"    ...
6d9335d0200b   docker/whalesay    "cowsay tutego"           ...
```

To prevent data in a container from being lost when it’s deleted, it can be stored on the host system using *data volumes* (*volumes*, for short) on the host system. The general syntax for creating a container with volumes is as follows:

```
docker run -v [HOST_PATH]:[CONTAINER_PATH] [IMAGE_NAME]
docker run --mount "type=volume,source=[HOST_PATH],target=[CONTAINER_PATH]"
[IMAGE_NAME]
```

To ensure that the data from the H2 database can be accessed outside the container, we need to make one final adjustment before starting H2. The database can be launched using the directory where the database is currently located (in my case, this is the user directory located at *C:\Users\christian*):

```
$ docker run -d -p 9092:9092 -p 8082:8082 -v C:\Users\christian:/h2-
data thomseno/h2
```

Our Spring application could access the H2 database in the Docker container with the JDBC URL `jdbc:h2:tcp://localhost/./unicorns`, and the data is stored at the host.

11.2.4 Prepare a Spring Boot Docker Application

We’ve only obtained an image from Docker Hub so far, but your Java application can also be packaged into an OCI image. For a simple demonstration, we’ll use this small program with a REST endpoint that accesses the database via the `JdbcTemplate`:

```
@SpringBootApplication
public class SpringBootDockerApplication {
  @Bean
  RouterFunction<ServerResponse> routes( JdbcTemplate jdbc ) {
   var sql = "SELECT count(*) FROM profile";
   return route().GET( "/api/stat/total",
                     __ -> ok().body(jdbc.queryForObject(sql, String.class)))
                 .build();
  }

  public static void main( String[] args ) {
    SpringApplication.run( SpringBootDockerApplication.class, args );
  }
}
```

To shorten the example as much as possible, the program uses a `RouterFunction` instead of `@RestController`. This defines a "route" on `/api/stat/total`. A `GET` call to the endpoint will access the database and start a small query.

There are several ways to create OCI container images for Spring programs, each with its advantages and disadvantages, as follows:

- **Dockerfile**
  This approach involves creating a custom Dockerfile that defines the necessary steps to build the image. While this method provides flexibility, it can also be time-consuming and requires knowledge of Docker.
- **Spring Boot plugins**
  Spring Boot provides plugins that can be used to build and package applications into a Docker image. This method is straightforward and doesn't require knowledge of Docker, but it may not be as flexible as the Dockerfile approach.
- **Other plugins**
  Other plugins include the Spotify Maven plugin, Palantir Gradle plugin, Google Jib Maven plugin, and Gradle plugins.

Spring's official website, *https://spring.io/guides/topicals/spring-boot-docker/*, provides a comparison of these solutions to help developers choose the best option for their needs.

Set an Image Name

The plugin goal offered by Spring Boot is simple, and we don't have to configure anything, so let's go this route. Before we start, however, we want to explicitly set the name of the image. Of course, every image has a default name, but this way, you can give it a shorter name. In the Spring Boot Maven plugin, a configuration block is inserted for this purpose:

```
<plugin>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-maven-plugin</artifactId>
  <configuration>
    <image>
      <name>tutego.date4u</name>
    </image>
  </configuration>
</plugin>
```

spring-boot:build-image

To create an OCI image, we call a plugin goal from the Spring Boot Maven plugin:

```
$ mvn -DskipTests spring-boot:build-image
```

Specifying with `-DskipTests` ignores the test cases, and the build is slightly faster.

The plugin goal `spring-boot:build-image` uses the local Docker daemon, and a separate Dockerfile isn't necessary. When building, the plugin falls back on a *cloud-native buildpack* (*https://buildpacks.io*). It takes longer the first time it's generated, but you end up with an OCI image.

You can see this with `docker images`:

```
$ docker images
REPOSITORY              TAG             IMAGE ID           CREATED         SIZE
paketobuildpacks/run    base-cnb        523e5e0ad089    7 days ago      87.2MB
gcr.io/paketo-buildpacks/↩
builder          base-platform-api-0.3  469751807bc6  40 years ago       587MB
spring-boot-docker      0.0.1-SNAPSHOT  a421598b9d7b  40 years ago       257MB
```

We could start our program, but this won't work completely because the database isn't set. If you're still curious, you can write the following:

```
$ docker run -it -p 8080:8080 spring-boot-docker:0.0.1-SNAPSHOT
```

At *http://localhost:8080/api/stat/total*, you can reach the web server, but the browser will display a *Whitelabel error page* because the database can't be reached. You'll be able to see the application errors in the console, and you can also interact with the application from the console. This is provided by the options `-it` (*keep STDIN open even if not attached*) and `-t` (*allocate a pseudo-tty*).

At least we know that the server is running. H2 has also built an in-memory database, but we want to set our own connection data in the next step. Actually, the H2 database is already running too (at least that's what we started before), but it's running in a container, and that's completely isolated from the Date4u application. So, we need to combine both containers into some kind of virtual network. Furthermore, the H2 database *must* run before the Spring application, and this should happen in an automated way.

11.2.5 Docker Compose

When creating a software application, it's often necessary to incorporate many other components, such as databases, messaging systems, and email servers. All these services need to be coordinated and started in a specific order. Docker has a tool called *Docker Compose* that addresses this need. Docker Compose allows multiple containers to be started and dependencies to be defined. With Docker Compose, you define an application stack in the form of a YAML Ain't Markup Language (YAML) file. This file describes all the containers required for the application; Docker Compose works through the stack to ensure that the containers are started in the correct order, and restarts any that fail.

Docker Compose is installed together with Docker Desktop. On the command line, you can test the following:

```
$ docker-compose version
Docker Compose version v2.12.0
```

Docker Compose File

The YAML file for Docker Compose is also called a *composition*—and it can look like this:

```
version: "3.8"

services:
  database:
    image: thomseno/h2
    container_name: db
    ports:
      - '9092:9092'
      - '8082:8082'
    volumes:
      - 'C:\Users\christian:/h2-data'
  application:
    image: tutego.date4u:latest
    container_name: date4u-server
    restart: always
    ports:
      - "8080:8080"
    depends_on:
      - database
    environment:
      - SPRING_DATASOURCE_URL=jdbc:h2:tcp://db/./unicorns
      - SPRING_DATASOURCE_USERNAME=user
      - SPRING_DATASOURCE_PASSWORD=pass
```

In a Docker Compose file, the first line always specifies the version identifier, which is currently `3.8`. Following that, the file defines services, each with a name and an associated image. The names are arbitrary and, in the example YAML file, two services are defined: `database` or the H2 database and `application` for the main program.

In addition to the image, services can also define dependencies, environment variables, working directories, volumes, or port forwarding. In the provided example, the H2 database needs to share its ports with the outside world and has a working directory, so it requires a volume. The demo Spring application doesn't need a volume; however, the real Date4u application, which stores images in the file system, does need one.

The Spring application depends on the database, which is specified using `depends_on`. The settings via environment variables are used to set the JDBC connection data. When Docker Compose is run, it works through the application stack and ensures that everything is in the correct order, including dependencies, and performs any necessary restarts.

The application stack is started by the following command:

```
$ docker-compose up
```

With this, Docker Compose starts the database and the Spring application, and access to the endpoint succeeds.

11.2.6 Terminate Applications with an Actuator Endpoint

When running applications in a Docker container, they need to be shut down correctly. If dependency `org.springframework.boot:spring-boot-starter-actuator` exists, there is a special endpoint `/actuator/shutdown`, which can be used to shut down the container in a controlled way.

The endpoint isn't enabled by default—that would be risky—so it must be enabled, for example, in *application.properties*:

```
# enable shutdown endpoint
endpoints.shutdown.enabled=true
# no default security
endpoints.shutdown.sensitive=false
```

When you call the endpoint from outside, the Spring Boot application shuts down automatically.

11.3 Summary

In this chapter, we've explored the process of assembling our application into an archive and running it on a target environment. However, in recent years, there has been a shift toward deploying applications in the cloud. Therefore, it's essential to understand deployment options in the cloud, such as *OCI containers*, *Kubernetes*, and platforms such as *Red Hat OpenShift*.

Software development in the cloud often involves checking the source code into a version control system, which is already a part of the cloud infrastructure. The build system then performs a checkout, runs the tests, and prepares the software for deployment. The objective is to automate the build process to the maximum extent possible so that new deployments can be done quickly and easily.

Appendix A

Migration from Spring Boot 2 to Spring Boot 3

Spring Framework and Spring Boot have been around for a while and have gone through several changes over the years. For instance, when transitioning from Spring Boot 1 to Spring Boot 2, adjustments had to be made to repositories, configuration properties, and Spring Boot Actuator project paths. However, that was a while ago. Moving from Spring Boot 2 to Spring Boot 3 also requires some modifications. This appendix provides a brief outline of the migration process.

A.1 Preparation

When migrating, you likely won't migrate directly from an old version to the latest one. The usual approach is incremental. That is, before moving to Spring Boot 3, the team should move to the latest Spring Boot 2 release. This was version 2.7 when Spring Boot 3 was introduced.

Spring Boot 2.7 prepares many things, so those using Spring Boot 2.7 will have an easy time moving to Spring Boot 3. From Spring Boot 2.6 to 2.7, there were quite a few updates on how web security is configured.

It's crucial to note that Spring Framework 6 and, as a result, Spring Boot 3, requires Java 17. If you're currently running your application with Java 8, you should compile and test it with Java 17.

In summary, before thinking about Spring Boot 3, you should first update your Spring Boot version and then fully test the program with a current Java runtime environment.

A.2 Jakarta EE 9

One of the most significant changes to the Spring Framework concerns the transition to Jakarta Enterprise Edition (Jakarta EE) 9. This is associated with the change of the root package. What used to be available under `javax` as Java Extension is now the root package `jakarta` in Jakarta EE. This includes a whole range of products:

- Jakarta Annotations
- Jakarta Bean Validation
- Jakarta Mail

- Jakarta Messaging
- Jakarta Persistence
- Jakarta Servlet
- Jakarta XML Binding

There are different approaches to the migration. One variant is to change the dependencies and then let the IDE correct the broken `import` declarations. Another approach is to work classically with search and replace.

Tip

If you use IntelliJ, you can use **Migrate Packages and Classes** to convert packages automatically. The IntelliJ website at *www.jetbrains.com/idea/guide/tutorials/migrating-javax-jakarta/use-migration-tool* documents the procedure.

A.3 Other Innovations

Spring Boot 3 provides a few new features, but they aren't visible if you're already using Spring Boot 2.7. A summary can be found on the referenced subpages at *https://github.com/spring-projects/spring-boot/wiki/Spring-Boot-3.0-Release-Notes*.

However, there are two major issues where the hoped-for breakthrough wasn't achieved:

- One of the big topics of Spring Framework version 6 and Spring Boot 3 was native compilation. Several years ago, the *Spring Native*[1] project was started to explore the possibilities. Later, the components were integrated into Spring Boot 3. However, the Spring team realized that native compilation is harder than expected after all, and the journey is still far from over. It remains a work in progress and should not be considered finished. However, all core models such as Spring Data and Spring Web MVC are supported right now.
- The Spring team also wanted to focus on the Java platform module system. However, this hardly plays a role in practice and was therefore given little priority.

Virtual threads realized via *Project Loom* (*https://openjdk.org/projects/loom*), are a new topic. However, as of Java 20, virtual threads aren't part of the Java standard library, but only available in a preview mode.[2] Although they play no role in the Spring Framework the what impact they could have has been discussed.[3]

---

[1] *https://github.com/spring-projects-experimental/spring-native*

[2] *https://openjdk.org/projects/jdk/19/*

[3] *https://spring.io/blog/2022/10/11/embracing-virtual-threads*

A.4 Spring-Boot-Properties-Migrator

Migrating an application to Spring Boot 3 basically affects two areas: changed types or methods, and configuration properties. Sporadically, settings are deleted or renamed. The Spring Boot team documents changes in a change log, which is definitely worth a look (see, e.g., the first milestone: *https://github.com/spring-projects/spring-boot/wiki/Spring-Boot-3.0.0-M1-Configuration-Changelog*).

To ensure that the removal or name change of a configuration property doesn't lead to the failure of specific components, the Spring team has developed the `spring-boot-properties-migrator`.

In the first step, a dependency is integrated:

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-properties-migrator</artifactId>
  <scope>runtime</scope>
</dependency>
```

Then, when you build and run the product, Spring Boot examines itself and shows which properties have been deleted or renamed. This is easy to see in the log stream; the messages come from the `PropertiesMigrationListener`. After the migration, the dependency is deleted again.

A.5 Spring Boot Migrator Project

Many customizations could be automated. The Spring team has released an experimental project called *Spring Boot Migrator* (SBM; see *https://github.com/spring-projects-experimental/spring-boot-migrator*). It can always be used with a project managed via Maven that uses at least Java version 11.

To use the tool, you first download a Java archive from the website and then run it:

```
$ java -jar spring-boot-migrator.jar
migrator:> scan C:/Users/christian/Desktop/date4u/main
```

The `scan` command scans the project, and `apply` can trigger a migration. The conversion isn't guaranteed to work smoothly—this project is also a work in progress.

Internally, SBM is based on the *OpenRewrite* project (*https://github.com/openrewrite/rewrite*). Rules are set so that certain patterns in the source code are recognized and replaced by other patterns.

A.6 Dependency Upgrades

The Spring Boot projects have innumerable subdependencies, and the version numbers are managed by Spring Boot itself. Generally, it's enough to raise the version number of Spring Boot so that the managed projects automatically receive an update.

Because the root package name was changed for the Jakarta projects, an update isn't backward compatible. Therefore, the version numbers between Spring Boot 2 and Spring Boot 3 show large jumps (see Table A.1).

Dependency	Version under Spring Boot	
	2.7	3.0
`jakarta.activation:jakarta.activation-api`	1.2.*	2.1.*
`jakarta.annotation:jakarta.annotation-api`	1.3.*	2.1.*
`jakarta.jms:jakarta.jms-api`	2.0.*	3.1.*
`jakarta.json:jakarta.json-api`	1.1.*	2.1.*
`jakarta.json.bind:jakarta.json.bind-api`	1.0.*	3.0.*
`jakarta.mail:jakarta.mail-api`	1.6.*	2.1.*
`jakarta.persistence:jakarta.persistence-api`	2.2.*	3.1.*
`jakarta.servlet:jakarta.servlet-api`	4.0.*	6.0.*
`jakarta.servlet.jsp.jstl:jakarta.servlet.jsp.jstl-api`	1.2.*	3.0.*
`jakarta.transaction:jakarta.transaction-api`	1.3.*	2.0.*
`jakarta.validation:jakarta.validation-api`	2.0.*	3.0.*
`jakarta.websocket:jakarta.websocket-api`	1.1.*	2.1.*
`jakarta.ws.rs:jakarta.ws.rs-api`	2.1.*	3.1.*
`jakarta.xml.bind:jakarta.xml.bind-api`	2.3.*	4.0.*
`jakarta.xml.soap:jakarta.xml.soap-api`	1.4.*	3.0.*
`jakarta.xml.ws:jakarta.xml.ws-api`	2.3.*	4.0.*

Table A.1 Selected Versions of Spring Boot 2.7 and Spring Boot 3.0

However, Spring Boot 2 doesn't use the current major or minor releases for many other subprojects either. Table A.2 gives a few examples.[4]

---

[4] Extracted from *https://mvnrepository.com/artifact/org.springframework.boot/spring-boot-dependencies/2.7.7*.

Dependency	Version under Spring Boot	
	2.7	3.0
`ch.qos.logback:logback-core`	1.2.*	1.4.*
`com.fasterxml.jackson.core:jackson-databind`	2.13.*	2.14.*
`com.github.ben-manes.caffeine:caffeine`	2.9.*	3.1.*
`com.google.code.gson:gson`	2.9.*	2.10
`com.zaxxer:HikariCP`	4.0.*	5.0.*
`io.reactivex:rxjava`	1.3.*	2.2.*
`io.undertow:undertow-core`	2.2.*	2.3.*
`net.sf.ehcache:ehcache`	2.10.*	3.10.*
`org.apache.tomcat.embed:tomcat-embed-core`	9.0.*	10.1.*
`org.codehaus.groovy:groovy`	3.0.*	4.0.*
`org.elasticsearch:elasticsearch`	7.17.*	8.5.*
`org.flywaydb:flyway-core`	8.5.*	9.7.*
`org.hibernate.validator:hibernate-validator`	6.2.*	8.0.*
`org.hibernate.validator:hibernate-validator`	6.2.*	8.0.*
`org.hibernate:hibernate-core`	5.6.*	6.1.*
`org.jooq:jooq`	3.14.*	3.17.*
`org.liquibase:liquibase-core`	4.9.*	4.17.*
`org.mongodb:mongodb-driver-core`	4.6.*	4.7.*
`org.slf4j:slf4j-api`	1.7.*	2.0.*

Table A.2 Selected Versions of Spring Boot 2.7 and Spring Boot 3.0 for Subprojects

There are also adaptations for test libraries (see Table A.3).

Dependency	Version under Spring Boot	
	2.7	3.0
`org.junit:junit-bom`	5.8.*	5.9.*
`org.mockito:mockito-bom`	4.5.*	4.8.*

Table A.3 Selected Versions of Spring Boot 2.7 and Spring Boot 3.0 for Test Libraries

The change from Mockito 4.5 to 4.8 means an important change in the annotation attribute `lenient`. So, from

```
@Mock( lenient = true )
```

the following must become

```
@Mock( strictness = Strictness.LENIENT )
```

So far, however, `lenient` is only `deprecated`, not deleted.

Appendix B
The Author

Christian Ullenboom is an Oracle-certified Java programmer and has been a trainer and consultant for Java technologies and object-oriented analysis and design since 1997.

Index

_id ... 700
[.yml] ... 224
{#entityName} ... 621
@Access ... 538
@ActiveProfiles ... 310
@AliasFor ... 128
@ApiResponses ... 822
@Async ... 362
@AttributeOverride ... 552
@AutoConfigureTestDatabase ... 685
@Basic ... 539
@Cacheable ... 345–346
@CacheConfig ... 358
@CacheEvict ... 351
@CachePut ... 350
@Check, Hibernate ... 540
@Column ... 539
@ConditionalOnBean ... 173
@ConditionalOnClass ... 173
@ConditionalOnCloudPlatform ... 173
@ConditionalOnDefaultWebSecurity ... 844
@ConditionalOnExpression ... 173
@ConditionalOnJava ... 173
@ConditionalOnJndi ... 173
@ConditionalOnMissingBean ... 173
@ConditionalOnMissingClass ... 173
@ConditionalOnNotWebApplication ... 173
@ConditionalOnProperty ... 173
@ConditionalOnResource ... 173
@ConditionalOnSingleCandidate ... 173
@ConditionalOnWarDeployment ... 174
@ConditionalOnWebApplication ... 173
@Container ... 689
@Controller ... 734
@ControllerAdvice ... 788
@Convert ... 546
@Converts ... 546
@CreatedBy ... 672
@CreatedDate ... 672
@DataElasticsearchTest ... 718
@DataJpaTest ... 683
@DataSizeUnit ... 215
@DateTimeFormat ... 768
@DependsOn ... 146
@Document ... 701
@DurationUnit ... 216
@ElementCollection ... 553
@Embeddable ... 551
@Embeddable, Schlüssel ... 550
@Embedded ... 551
@EmbeddedId ... 550
@EnableAsync ... 362
@EnableCaching ... 345
@EnableFeignClients ... 875
@EnableJpaAuditing ... 672
@EnableRetry ... 390
@Entity ... 537
@EntityListeners ... 672–673
@Enumerated ... 543
@ExceptionHandler ... 783
@GeneratedValue ... 489, 548
@GetExchange ... 875
@Hidden ... 820
@HttpExchange ... 875
@Id ... 489, 547
@Id, Spring Data Commons ... 700
@IdClass ... 550
@InitBinder ... 773
@InjectMocks ... 301
@JoinColumn ... 557
@JsonIgnore ... 796
@JsonMixin ... 748
@JsonTest ... 319
@LastModifiedBy ... 672
@LastModifiedDate ... 672
@Lob ... 544
@LocalServerPort ... 831
@ManyToMany ... 567
@ManyToOne ... 562
@MappedSuperclass ... 553
@Mock ... 300
@MockitoSettings ... 301
@Name ... 169
@NamedQueries ... 518
@NamedQuery ... 517
@OneToOne ... 557
@Operation ... 822
@OrderBy ... 565
@PageableDefault ... 813
@PathVariable ... 760
@PersistenceContext ... 493
@PostConstruct ... 151
@PreDestroy ... 151
@Procedure ... 627
@Qualifier ... 169
@Query ... 615

@Recover ... 392
@RequestHeader ... 765
@ResponseBody ... 735
@ResponseStatus ... 753, 780
@Retryable ... 389
@Scope ... 169
@ServiceConnection ... 689
@ServletComponentScan ... 731
@Singleton ... 169
@SortDefault ... 814
@SortDefaults ... 814
@SpringBootApplication ... 78
@Spy ... 301
@Table ... 536
@Testcontainers ... 689
@Transactional ... 471
@Transient ... 539
@Valid ... 381
@Validated ... 381
@WebMvcTest ... 827
@WebServlet ... 731
/META-INF/resources ... 725
/public ... 725
/resources ... 725
/static ... 725
401, status code ... 843
443, port ... 728
8080, port ... 724
8443, port ... 728

A

ABS, JPQL ... 509
AbstractAuditable ... 555
AbstractInterruptibleBatchPreparedStatementSetter ... 446
Abstractions ... 127
AbstractLobStreamingResultSetExtractor ... 455
AbstractPersistable ... 555
AbstractThrowableAssert ... 293
AccessType ... 538
ACID principle ... 460
Actuator endpoints ... 886
Aggregate ... 655
Aggregate root ... 655
Ahead-of-time compilation ... 42
Aliases ... 73
Alias name ... 127
Anemic domain model ... 485
Annotation processor ... 58, 649
Ant-Style pattern ... 265
Apache HTTP Server ... 721
Apache Solr ... 710
Apache Tika ... 709
API Blueprint ... 817
application/json ... 752
application/xml ... 752
Application context ... 70
ApplicationContextAware ... 156
ApplicationConversionService ... 267
ApplicationEventMulticaster ... 258
ApplicationEventPublisher ... 252
ApplicationRunner ... 238
Application server ... 41
Arrays and lists ... 211
Artificial key ... 547
ASC, ORDER BY, JPQL ... 510
AsciiDoc ... 824
Assert ... 330
AssertJ ... 289, 385
assertThatThrownBy(...) ... 293
AsyncConfigurer ... 372
AttributConverter ... 544
Auditing ... 672
AuditingEntityListener ... 673
AuditorAware ... 673
Authentication ... 837–838
AuthenticationManager ... 842
AuthenticationProvider ... 846
Authorization ... 837
Auto-commit mode ... 461
Auto-configuration ... 38, 73, 170
Automatic dirty checking ... 534
autowireCandidate ... 154
AutowiredAnnotationBeanPostProcessor ... 161
Aware ... 157

B

Banner ... 883
BaseJavaMigration ... 680
Basic attributes ... 542
BasicAuthenticationFilter ... 856
Batch operation ... 443
BatchPreparedStatementSetter ... 444
Batch updates ... 531
BeanClassLoaderAware ... 156
BeanDefinition ... 158
BeanDefinitionBuilder ... 158
BeanNameAware ... 156
BeanPostProcessor ... 159
BeanPropertyRowMapper ... 430
BETWEEN, JPQL ... 507
Bidirectional relationships, entity beans ... 559

Bill of materials (BOM) ... 55
Binary large objects ... 451, 544
BindingResult ... 281
BooleanBuilder ... 653
BooleanExpression ... 653
BSON ... 700
Builder pattern ... 69
ByteArrayHttpMessageConverter ... 746
Bytecode-enhanced dirty tracking ... 534

C

Cache, distributed vs. local ... 359
Cache implementations ... 358
CacheManager ... 361
cacheNames, @CacheConfig ... 358
Caching ... 343
Caffeine ... 359
Calender versioning ... 837
CamelCaseToUnderscoresNamingStrategy ... 541
CapturedOutput ... 885
Cardinality, database ... 560
CascadeType ... 574
Cascading ... 573
Certificate authority ... 726
cglib ... 340
Chain of responsibility design pattern ... 847
Character large objects ... 451, 544
Chart.js ... 750
Classpath ... 181
Clauses, Criteria API ... 640
Cloud-native buildpack ... 912
Code-first approach ... 485
Collection, MongoDB ... 695
Collection, RESTful API ... 795
CommandLineRunner ... 238
Common annotations for the Java Platform ... 139
Common Gateway Interface ... 729
Commons Logging API ... 62
Compass application ... 697
CompletableFuture ... 365
Composition ... 913
Compound primary key ... 549
CONCAT, JPQL ... 508
condition ... 348
ConfigDataLocationNotFoundException ... 224
Config folder ... 221
Config tree ... 224
Configuration classes ... 111
ConnectionCallback ... 442
Connection tables ... 566
ConstraintViolationException ... 381
Constructor expression, JPQL ... 516
Constructor expressions ... 665
Constructor injection ... 98
Containerization ... 904
Container-managed object ... 410
Container-managed persistence (CMP) ... 479
Containers ... 904
Content negotiation ... 752
Content type ... 741
Controller, web ... 733
Controller advice ... 787
ConversionService ... 196, 266
Core-starter ... 60
createNamedQuery(...) ... 517
createQuery(...) ... 500
Criteria ... 703
Criteria, Spring Data JDBC ... 658
Criteria API ... 635
CriteriaBuilder ... 637
CriteriaDefinition ... 658
CriteriaQuery ... 637
Cronjob ... 326
Cross-cutting concern ... 470
CRUD operation ... 408, 580
CrudRepository ... 589–590
CURRENT_DATE ... 510
CURRENT_TIME ... 510
CURRENT_TIMESTAMP ... 510

D

DaoAuthenticationProvider ... 847
data.sql ... 690
DataAccessException ... 416
Data access objects ... 579
Database as a service ... 696
Database-first approach ... 485
DataBinder ... 279
DataClassRowMapper ... 431
Data Definition Language ... 483
DataSize ... 214
DataSourceAutoConfiguration ... 418
DataSourceBuilder ... 418, 420
DataSourceProperties ... 419
DataSourceUtils ... 421, 464
Data transfer object (DTO) ... 665, 797
DataUnit ... 215
Data volumes, Docker ... 910
Declarative ... 76
DefaultConversionService ... 267

DefaultListableBeanFactory ... 157
DefaultLobHandler ... 453
Default profile ... 232
Default web security ... 843
DeferredResult ... 809
Dependency ... 50
Dependency injection ... 97
Derived query method ... 622, 628, 705
DESC, ORDER BY, JPQL ... 510
destroyMethod ... 154
Destroy method inference ... 154
Dev tools ... 739
Diff-based, dirty-checking-stategy ... 534
Digital certificate ... 726
Directory traversal ... 293
DirtiesContext ... 313
Dispatcher servlet ... 733
DisposableBean ... 155
DISTINCT, JPQL ... 512
Distributed transactions ... 461
Docker ... 906
Docker Compose ... 913
Docker Desktop ... 906
Dockerfile ... 907
Docker Hub ... 907
Document-oriented NoSQL databases ... 694
Domain, JPQL ... 498
Dry-run, Flyway ... 681
Dynamic proxies ... 340
Dynamic schema ... 695

E

Eager loading ... 568
Eclipse Adoptium ... 27, 711
Eclipse IDE ... 48
EclipseLink JPA ... 480
Elasticsearch ... 707
ElasticsearchContainer ... 719
Embedded MongoDB ... 706
Enabled, Spring Actuator ... 887
EnableScheduling ... 325
Enterprise JavaBeans 3.0 ... 479
Entity beans ... 408
Entity graph ... 571
EntityManager API ... 481
EnumType ... 543
EnvironmentAware ... 156
Errorlevel ... 241
Eventbus ... 252
Evolutionary database design ... 676
Exceptions ... 372

execute(...), JdbcTemplate ... 422
Execute-around idiom ... 466
ExecutionException ... 366
Exit code ... 241
ExitCodeGenerator ... 242
Exposed, Spring Actuator ... 888
Expression, Criteria API ... 640
ExpressionParser ... 185
Extension Hint ... 224
EXTRACT, JPQL-Funktion ... 510

F

Factory methods ... 111
FailureAnalyzer ... 106
Fallback match ... 138
Fallback method, recover ... 393
FetchType ... 568
Field, document ... 694
Field, Elasticsearch ... 713
Field-based access ... 537
Field injection ... 98, 100
FIGlet ... 883
Fire-and-forget strategy ... 362
Fluent API ... 69
Flush, Jakarta Persistence ... 530
Flyway ... 625, 676
Fragment interface ... 663
Front controller ... 733
Fuzzy search ... 707

G

Gauges ... 896
GenericPropertyMatcher ... 613
getReference() ... 496
getResultList() ... 500
Global transactions ... 461
GraalVM Native Image ... 42
Gradle ... 43
Granted authorities ... 838
GROUP BY, JPQL ... 515
Gson ... 751

H

H2, Datenbank ... 409
H2 Console ... 402
H2 database ... 397
Hamcrest ... 289
Handler mapping ... 733
Handler method ... 734

HEAD, HTTP ... 792
Headless mode ... 69
HealthIndicator ... 893
Hibernate Envers ... 674
Hibernate Jpamodelgen ... 643
Hibernate ORM ... 480
Hibernate Query Language ... 499
Hibernate Validator ... 376
Hierarchical contexts ... 165
HikariCP ... 413
Hint, Query ... 572
Hot code swapping ... 738
HttpEntity ... 765
HTTP-header ... 740
HttpHeaders ... 765
HttpMessageConverter ... 746
HttpServlet ... 730
HttpServletRequest ... 731
HttpServletResponse ... 731
HttpStatus ... 753
HTTP status code ... 741
Hypermedia as the engine of application state ... 790

I

ID class ... 549
Idempotency ... 791
Identification variable ... 498
ID object ... 549
Image ... 905
Image specification ... 905
Impedance mismatch ... 477
Implicit mapping ... 489
IN, JPQL ... 507, 617
IncorrectResultSizeDataAccessException ... 423
Indexing ... 708
Index parameter
  @Query ... 616
  JPQL ... 503
Infobip Spring Data Querydsl ... 654
Initial configuration ... 68
InitializingBean ... 155
Initializr ... 43, 67
initMethod ... 154
InjectionPoint ... 122
InMemoryUserDetailManager ... 849
Integration tests ... 682
IntelliJ IDEA ... 45, 49
IntelliJ Ultimate Edition ... 190, 378, 632
Interface 21 ... 34
Interface Segregation Principle (ISP) ... 661
Internationalization ... 283
InvalidDataAccessApiUsageException ... 424
Invasive technologies ... 34
Inversion of control ... 97, 197
Inverted index ... 709
IS EMPTY, JPQL ... 566

J

Jackson ... 315, 751
Jackson2ObjectMapperBuilder ... 317
Jaeger ... 900
jakarta.persistence.fetchgraph ... 573
Jakarta Annotations API ... 151
Jakarta Bean Validation ... 41, 375
Jakarta Enterprise Beans ... 34
Jakarta Faces ... 732
Jakarta Persistence ... 479
Jakarta Persistence API ... 41, 590
Jakarta Persistence Criteria API ... 635
Jakarta Persistence Query Language ... 481, 600, 705
Jakarta Persistence specification ... 481
Jakarta RESTful Web Services ... 382, 732
Jakarta XML Binding ... 751
Java 2 Platform, Enterprise Edition ... 34
Java configuration class ... 82
Java Enterprise Edition ... 40
Java Management Extensions (JMX) ... 886
JavaMigration ... 680
Java Persistence API ... 479
Java Platform, Enterprise Edition ... 34
Java Platform, Standard Edition ... 33
Java virtual machine (JVM) ... 739
JAXB ... 751
Jaxb2RootElementHttpMessageConverter ... 746
JCache ... 345
JdbcAggregateTemplate ... 657
JdbcTemplate ... 411
JdbcUserDetailsManager ... 850
JDK dynamic proxy ... 340
Jetty ... 724
Jimfs ... 304
JLine ... 114
Join table ... 566
jOOQ ... 647
JPA Buddy ... 492
JpaRepository ... 596
JpaSpecificationExecutor ... 638
JSON ... 314, 889
JSONassert ... 289
JSON-B ... 751

JsonContent ... 319
JsonDeserializer ... 323
JsonPath ... 289
JsonSerializer ... 323
JSON Web Token ... 859
JSR 250 ... 139
JUnit 5 ... 289
JwtClaimsSet ... 865
JwtDecoder ... 863
JwtEncoder ... 863

K

Kebab format ... 194
key, @Cacheable ... 355
KeyGenerator ... 352
keyGenerator, @Cacheable ... 357
KeyHolder ... 436
keytool ... 727
Kotlin ... 43

L

Lazy ... 334, 606
Lazy initialization ... 148
Lazy loading ... 567
LdapUserDetailsManager ... 850
LENGTH, JPQL ... 508
LIMIT, SQL ... 501
Liquibase ... 676
ListCrudRepository ... 594
ListPagingAndSortingRepository ... 599
Lite-Bean ... 112, 120
LobHandler ... 453
LOCAL DATE ... 510
LOCAL DATETIME ... 510
Locale, Java SE ... 283
Localization ... 283
LOCAL TIME ... 510
Local transactions ... 461
LOG_AND_PROPAGATE_ERROR_HANDLER ... 259
LOG_AND_SUPPRESS_ERROR_HANDLER ... 259
log4j2 ... 880
Logback ... 880
logging.group ... 880
logging.level ... 880
Logging facades ... 62
Log groups ... 880
Log level ... 64
Lombok ... 202, 383
LOWER, JPQL ... 508
Lucene ... 708

M

Mail API ... 41
management.endpoint.health.show-components ... 892
management.endpoint.shutdown.enabled ... 888
management.endpoints.web.exposure.exclude ... 889
mappedBy ... 560
MappingJackson2HttpMessageConverter ... 746
MappingSqlQuery ... 458
MapSqlParameterSource ... 440
MapStruct ... 834
Marshalling, JSON ... 315
Maven ... 43, 903
  wrapper ... 45
Maven resource filtering ... 226
Meta-annotations ... 78
Metamodel ... 643
MethodInterceptor ... 342
Metrics ... 894
Micrometer ... 895
Micrometer Tracing ... 900
Migrations ... 677
Migration scripts ... 676
Milestones ... 58
MIME type ... 741
MockBean ... 312
MockHttpServletRequestBuilder ... 828
Mockito ... 289, 298
MockitoExtension ... 300
MockMvc ... 827
MockMvcRequestBuilders ... 828
MockMvcResultMatchers ... 828
MOD, JPQL ... 509
Model-view-controller (MVC) ... 191
MongoClient ... 699
MongoDB ... 694
MongoDB Atlas ... 696
MongoRepository ... 704
MongoTemplate ... 701
Mono ... 868
Multi-document files ... 233
Multipart/form data ... 763
Multithreaded ... 730
MultiValueMap ... 330, 758
MutablePropertyValues ... 280

N

N + 1 select problem ... 571
Named parameter, @Query ... 616
NamedParameterJdbcTemplate ... 438
Named queries ... 498, 517
Native compilation ... 41
Natural keys ... 547
Near, derived query methods ... 706
nginx ... 721
Nimbus ... 863
NimbusJwtDecoder ... 866
NimbusJwtEncoder ... 864
NoOpCacheManager ... 361
NoOpPasswortEncoder ... 854
Normalization ... 709
NoSQL database ... 694
Nullable types ... 486

O

OAuth2ResourceServerConfigurer ... 865
ObjectId ... 700
ObjectMapper ... 317
Object-relational mapping ... 408, 475
OFFSET, SQL ... 501
Onion architecture ... 165
OpenAPI Generator ... 822, 876
openapi-generator-maven-plugin ... 823
OpenAPI specification ... 817
Open Container Initiative (OCI) ... 904
OpenFeign ... 873
OpenRewrite ... 919
Open session in view ... 570
Open Web Application Security Project
  (OWASP) ... 424
Optionals ... 331
OPTIONS, HTTP ... 792
Order ... 600
org.springframework.dao ... 416
org.springframework.jdbc ... 415–416
org.springframework.jdbc.core.simple ... 455
org.springframework.jdbc.object ... 458
org.springframework.jdbc.support ... 415
org.springframework.security.oauth2.jwt ... 863
orphanRemoval ... 577
Orphans ... 577
OutputCaptureExtension ... 885
Owning dide ... 557

P

Page ... 603
Pageable ... 602
Page-based pagination ... 607
PageRequest ... 602
Pagination ... 598
PagingAndSortingRepository ... 598
Pair ... 335
PasswordEncoder ... 852
Path matcher ... 743
PathPattern ... 743
Path variable ... 756, 760
Persistable ... 555
Persistence context ... 531
Persistence provider ... 480
Persistent attributes ... 538
Persistent fields ... 537
Persistent properties ... 537
Persistent unit file ... 531
Petclinic ... 36
PID ... 246
PKCS #12 ... 727
PlatformTransactionManager ... 462
Port mapping, Docker ... 909
Position parameters, @Query ... 616
Positionsparameter, JPQL ... 503
Predicate ... 637
Predicate, Querydsl ... 652
PreparedStatementCreator ... 435
ProblemDetail ... 786
Process identifier ... 246
Profile expression ... 237
Profiles ... 231
Projection ... 664
Project Loom ... 918
Prometheus ... 898
PromQL ... 898
PropertiesMigrationListener ... 919
Property-based access ... 537
Property editors ... 214
Property sources ... 193, 218
PropertySourcesPlaceholderConfigurer ... 225
PropertyValues ... 281
PropertyValueTransformer ... 613
Prototype ... 124, 168
Provider ... 169
ProviderManager ... 846
Proximity search ... 708
proxyBeanMethods ... 121
ProxyFactory ... 342
Proxy object ... 337
Proxy pattern ... 337

Q

QSort 651
Q-types 650
Qualifications 127
Quarkus 41
Quartz 328
Query 500, 703
Query, Spring Data JDBC 658
QueryByExampleExecutor 609
Query documents, MongoDB 705
Querydsl 647
QuerydslBinderCustomizer 816
QuerydslPredicateExecutor 651
QuerydslRepositorySupport 651
queryForObject(...), JdbcTemplate 423, 425
Query string 756

R

RAML 817
Random User Generator 388
Raw password 853
RDBMS object 458
RdbmsOperation 458
ReflectionTestUtils 303
Relational database management system (RDBMS) 498, 693
Relaxed binding 217
Replication 696
Repository 579, 597
RequestEntity 765
RequestMethod 743
ResourceBundle, Java SE 283
ResourceHttpMessageConverter 746
Resource provider 461
ResponseStatusException 781
RESTful API 790
ResultSetExtractor 434
RetryCallback 394
RetryContext 394
RetryTemplate 394
RFC 6570 760
Richardson Maturity Model 790
Rod Johnson 34
Root 637
RowCallbackHandler 433
RowMapper 428
Runtime Specification 905

S

Sample 608
Scheduled 325
schema.sql 686
Schema freedom 700
SecurityContext 839
SecurityContextHolder 839
SecurityFilterChain 843
SELECT, JPQL 498
SELECT N+1 update problem 483
Self-extracting JAR 51
Self-signed certificate 726
Semantic annotations 80
Semantic Query Model 646
Semantic versioning 837
server.port 724
server.ssl.* 727
Server Side Public License (SSPL) 695
Servlet API 34
Servlet container 729
ServletRegistrationBean 731
ServletUriComponentsBuilder 806
Setter injection 98–99
Sharding 696
Shutdown hook 153
SimpleAsyncTaskExecutor 258
SimpleAsyncUncaughtExceptionHandler 373
SimpleJdbcInsert 456
SimpleJpaRepository 582
SimpleKey 353
SimpleKeyGenerator 352
Simple Logging Facade for Java (SLF4J) 879
Singleton, RESTful API 795
Singletons 737
SIZE(...), JPQL 566
SLF4J 62
SnakeYAML 60
Snapshot, dirty-checking-stategy 534
Snapshots repository 58
SonarSource 736
Sort 600
Soundex code 519
Specification 637
SpelExpressionParser 185
SpEL expressions 185
SPRING_APPLICATION_JSON 219
spring.application.json 219
spring.cache.type 360
spring.config.additional-location 223
spring.config.import 223
spring.config.location 223

spring.config.name ... 223
spring.data.mongodb.* ... 699
spring.data.web.pageable ... 814
spring.flayway.enabled ... 686
spring.jdbc.template.fetch-size ... 451
spring.jdbc.template.max-rows ... 451
spring.jdbc.template.query-timeout ... 451
spring.jpa.hibernate.ddl-auto ... 687
spring.jpa.properties.hibernate.criteria.literal_handling_mode ... 646
spring.main.log-startup-info ... 884
spring.output.ansi.enabled ... 884
spring.pid.fail-on-write-error ... 246
spring.security.user ... 849
spring.shell.interactive.enabled ... 308
SpringApplicationBuilder ... 69
spring-boot.run.arguments ... 219
Spring Boot Actuator ... 879, 885
Spring Boot CLI ... 48, 65
Spring Boot Configuration Annotation Processor ... 208
Spring Boot DataSource Decorator ... 414
spring-boot-maven-plugin ... 903
Spring Boot Migrator ... 919
Spring Boot Properties Migrator ... 919
spring-boot\run ... 902
Spring Boot Starter JDBC ... 408
Spring Boot Starter MongoDB ... 699
Spring Boot Starter parent ... 55
Spring Boot Starter Web ... 723
Spring Boot versions ... 38
SpringBootWebSecurityConfiguration ... 843
Spring Cloud Config ... 231
Spring Cloud Function project ... 901
Spring Cloud Gateway ... 876
Spring Data Commons ... 589, 714
Spring Data Commons package ... 584
Spring Data Envers ... 675
Spring Data JDBC ... 655
Spring Data MongoDB ... 630, 698
Spring Data Web Support ... 811
Spring Developer Tools ... 739
springdoc.api-docs.path ... 820
springdoc-openapi-starter-webmvc-ui ... 820
Spring Expression Language (SpEL) ... 185, 348
SpringExtension ... 307
Spring HATEOAS ... 790
Spring-managed beans ... 37, 67
Spring Native ... 918
Spring Reactive ... 867
Spring REST Docs ... 824
Spring Retry project ... 387
Spring Security ... 837, 893
Spring Shell ... 89, 496
Spring Test ... 289
Spring Tool Suite ... 48
Spring Web ... 723
Spring WebFlux ... 732
Spring Web MVC ... 723
sql-error-codes.xml ... 417
SQLException ... 417
SQLExceptionTranslator ... 417
SQL injection ... 424
SqlParameterSource ... 439
SqlParameterSourceUtils ... 450
SQRL, JPQL ... 509
SSL certificate ... 726
Starter ... 60
Stemming ... 709
Stereotype ... 80
Streamable ... 332
StreamUtils ... 330, 333
StringHttpMessageConverter ... 746
StringMatcher ... 612
StringUtils ... 330
Stubbing ... 299
Subobjects ... 212
SUBSTRING, JPQL ... 508
Swagger UI ... 821

T

Table ... 496
TableBuilder ... 496
TableModel ... 496
TableModelBuilder ... 496
Target object ... 337
TaskExecutor ... 369–370
TaskUtils ... 259
Term ... 708
Testcontainers ... 687
TestEntityManager ... 691
TestPropertySource ... 309
Test slices ... 314
TextCriteria ... 706
ThreadPoolExecutor ... 371
ThreadPoolTaskExecutor ... 371
TimeUnit ... 214
Tokenization ... 708
Tomcat ... 34
Tools Launcher ... 51
TransactionCallback ... 466
TransactionCallbackWithoutResult ... 467
Transact SQL ... 625

Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

TRIM, JPQL 508
Tuple 514
TypedSort 601
TypeQuery 500

U

Unauthorized, HTTP 843
UncaughtExceptionHandler 373
Undertow 724
Unit testing 682
Universally Unique Identifier (UUID) 116
Unless 349
update(...), JdbcTemplate 423
UPPER, JPQL 508
UriComponents 805
UriComponentsBuilder 805
URI template 760
URI variables 774
UserDetailsManager 850
UserDetailsService 847
UserDetailsServiceAutoConfiguration 848
UsernameNotFoundException 848
UsernamePasswordAuthenticationToken 840
UUID 116
UUID, primary key 547

V

Validator 378
verify(...) 302
View-ID 734
Virtualization 904
Virtual threads 918
Visual Studio Code 48

W

Web Application Context 733
WebAsyncTask 808
WebClient 868
WebDataBinder 773
WebEnvironment 831
WebJars 726
Web service 789
WebTestClient 832
Whitelabel error page 770, 912
Windows Subsystem for Linux (WSL) 906
Wiring 98
Within, derived query methods 706
Workspace 51

X

XML, RESTful API 751

Z

Zipkin 900

Service Pages

The following sections contain notes on how you can contact us.

Praise and Criticism

We hope that you enjoyed reading this book. If it met your expectations, please do recommend it. If you think there is room for improvement, please get in touch with the editor of the book: meaganw@rheinwerk-publishing.com. We welcome every suggestion for improvement but, of course, also any praise!

You can also share your reading experience via Twitter, Facebook, or email.

Supplements

If there are supplements available (sample code, exercise materials, lists, and so on), they will be provided in your online library and on the web catalog page for this book. You can directly navigate to this page using the following link: http://www.rheinwerk-computing.com/5764. Should we learn about typos that alter the meaning or content errors, we will provide a list with corrections there, too.

Technical Issues

If you experience technical issues with your e-book or e-book account at Rheinwerk Computing, please feel free to contact our reader service: support@rheinwerk-publishing.com.

About Us and Our Program

The website http://www.rheinwerk-computing.com provides detailed and first-hand information on our current publishing program. Here, you can also easily order all of our books and e-books. Information on Rheinwerk Publishing Inc. and additional contact options can also be found at http://www.rheinwerk-computing.com.

Legal Notes

This section contains the detailed and legally binding usage conditions for this e-book.

Copyright Note

This publication is protected by copyright in its entirety. All usage and exploitation rights are reserved by the author and Rheinwerk Publishing; in particular the right of reproduction and the right of distribution, be it in printed or electronic form.

Your Rights as a User

You are entitled to use this e-book for personal purposes only. In particular, you may print the e-book for personal use or copy it as long as you store this copy on a device that is solely and personally used by yourself. You are not entitled to any other usage or exploitation.

In particular, it is not permitted to forward electronic or printed copies to third parties. Furthermore, it is not permitted to distribute the e-book on the Internet, in intranets, or in any other way or make it available to third parties. Any public exhibition, other publication, or any reproduction of the e-book beyond personal use are expressly prohibited. The aforementioned does not only apply to the e-book in its entirety but also to parts thereof (e.g., charts, pictures, tables, sections of text).

Copyright notes, brands, and other legal reservations as well as the digital watermark may not be removed from the e-book.

Digital Watermark

This e-book copy contains a **digital watermark**, a signature that indicates which person may use this copy. If you, dear reader, are not this person, you are violating the copyright. So please refrain from using this e-book and inform us about this violation. A brief email to info@rheinwerk-publishing.com is sufficient. Thank you!

Trademarks

The common names, trade names, descriptions of goods, and so on used in this publication may be trademarks without special identification and subject to legal regulations as such.

All products mentioned in this book are registered or unregistered trademarks of their respective companies.

Limitation of Liability

Regardless of the care that has been taken in creating texts, figures, and programs, neither the publisher nor the author, editor, or translator assume any legal responsibility or any liability for possible errors and their consequences.